Universal Computing

W F McColl

Oxford University

Abstract. The field of scalable computing is being redefined by the emergence of low cost parallel servers based on standard commodity microprocessors and off-the-shelf networking technologies. Many high performance applications will, in future, be carried out on such systems. Other applications, with very demanding communications requirements, will continue to be run on more specialised and expensive supercomputer systems. Over the next few years we will see the growth of a large and diverse global parallel software industry similar to that which currently exists for sequential computing. The main goal of that industry will be to produce scalable programs which, in addition to being fully portable, will offer high performance, in a predictable way, on any general purpose parallel architecture. The BSP model provides a discipline for the design of universal programs of this kind.

1 Introduction

Over the last thirty years or so we have seen the emergence and development of a huge global software industry. A large part of that industry is concerned with the production of portable applications software for the wide variety of sequential computers which are currently in use, from personal computers to large mainframes. The roots of this success story can be traced back to the work of von Neumann in the 1940s and to the developments in the areas of high level languages and compilers which followed on from that work in the 1950s and early 1960s. The model of a general purpose sequential computer proposed by von Neumann has served as the basic framework for almost all of the sequential computers which have been produced from the late 1940s to the present time. The stability which the model has provided has been crucial to the growth of the software industry over the years.

Since the earliest days of computing it has been clear that, sooner or later, sequential computing would be superseded by parallel computing. This has not yet happened, despite the availability of numerous parallel machines and the insatiable demand for increased computing power. For parallel computing to become the normal form of computing we require a model which can play a similar role to that which the von Neumann model has played in sequential

* This work was supported in part by ESPRIT Basic Research Project 9072 - GEPPCOM (Foundations of General Purpose Parallel Computing). Address: Programming Research Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, England. Email: `mccoll@comlab.ox.ac.uk`

computing. The emergence of such a model would stimulate the development of a new parallel software industry, and provide a clear focus for future hardware developments. For a model to succeed in this role it must offer three fundamental properties:

scalability - the performance of software and hardware must be scalable from a single processor to several hundreds or thousands of processors.
portability - software must be able to run unchanged, with high performance, on any general purpose parallel architecture.
predictability - the performance of software on different architectures must be predictable in a straightforward way.

It should also, ideally, permit the correctness of parallel programs to be determined in a way which is not much more difficult than for sequential programs.

During the last thirty years, a large number of parallel models have been proposed. They include: SIMD parallelism, synchronous message passing, logic programming, graph reduction, dataflow, and various forms of cache-based virtual shared memory. None of these approaches has, however, achieved all three main requirements.

The BSP model, which we describe in this paper, is quite different from many of the approaches which have been proposed in the past. It decouples the two fundamental aspects of parallel computation - communication and synchronisation. This decoupling is the key to achieving universal applicability across the whole range of parallel architectures. Recent research on BSP algorithms, architectures and languages has demonstrated convincingly that the BSP model can achieve all three of the requirements mentioned above.

2 Parallel Architectures

In recent years, the pursuit of higher performance from computers has forced the introduction of parallel computing in many areas, particularly in scientific and engineering applications and in database systems and transaction processing. The transition from sequential to parallel has not, however, been without its problems. On most of the early commercial parallel machines produced in the 1980s, scalable parallel performance could only be achieved by carefully exploiting the particular architectural details of the machine. Besides being extremely tedious and time consuming in many cases, this usually resulted in parallel applications software which could not be easily adapted to run on other machines. In a world of rapidly changing and diverse parallel architectures, this architecture dependence of the parallel software was a major weakness and seriously inhibited the growth of the field.

Over the last few years the situation has improved somewhat. For a variety of technological and economic reasons, the various classes of parallel computer in use (distributed memory machines, shared memory multiprocessors, clusters of workstations) have been steadily becoming more and more alike. The economic advantages of using standard commodity components has been a major factor in this convergence. Other influential factors have been the need to efficiently support a single address space on distributed memory machines for ease of programming, and the need to replace buses by networks to achieve scalability in shared memory multiprocessors. These various pressures have acted to produce a rapid and significant evolutionary restructuring of the parallel computing industry.

There is now a growing consensus that for a combination of technological, commercial, and software reasons, we will see a steady evolution over the next few years towards a "standard" architectural model for scalable parallel computing. It is likely to consist of a collection of (workstation-like) processor-memory pairs connected by a communications network which can be used to efficiently support a global address space. As with all such successful models, there will be plenty of scope for the use of different designs and technologies to realise such systems in different forms depending on the cost and performance requirements sought.

The simplest, cheapest, and probably the most common architectures will be based on clusters of personal computers. The Intel Pentium Pro microprocessor is an example of a high volume, commodity microprocessor for the personal computer market which provides hardware support for multiprocessing. Using such commodity microprocessor components, various companies are now producing very low cost multiprocessor systems for use as parallel servers. Commodity networking technologies such as ATM and new scalable software systems for communications will allow these to be assembled into clusters which will provide a very low cost option for many high performance commercial, scientific and internet applications.

At the other end of the spectrum, there will continue to be a small group of companies producing very large, very powerful and very expensive parallel supercomputer systems for those with applications which require those computing resources. The CRAY T3D is a good example of such a system. It is a very powerful distributed memory architecture based on the DEC Alpha microprocessor. In addition to offering high bandwidth global communications, it has several specialised hardware mechanisms which enable it to efficiently support parallel programs which execute in a global address space. The mechanisms include hardware barrier synchronisation and direct remote memory access. The latter permits each processor to get a value directly from any remote memory location in the machine, and to put a value directly in any remote memory location. This is done in a way which avoids the performance penalties normally incurred in executing such operations on a distributed memory architecture, due to processor synchronisation and other unnecessary activities in the low level systems software.

The process of architectural convergence which has been described brings with it the hope that we can, over the next few years, establish parallel computing as the standard method of computing, and begin to see the growth of a large and diverse global parallel software industry similar to that which currently exists for sequential computing. The main goal of that industry will be to produce scalable programs which, in addition to being fully portable, will offer high performance, in a predictable way, on any general purpose parallel architecture. The BSP model provides a discipline for the design of universal programs of this kind.

3 The BSP Model

3.1 Supersteps

In architectural terms, the BSP model is essentially the standard model described above. A ***bulk synchronous parallel (BSP) computer*** [9, 12] consists of a set of processor-memory pairs, a global communications network, and a mechanism for the efficient barrier synchronisation of the processors. A BSP computer operates in the following way. A computation consists of a sequence of parallel ***supersteps***, where each superstep consists of a sequence of steps, followed by a barrier synchronisation at which point all data communications will be completed. During a superstep, each processor can perform a number of computation steps on values held locally at the start of the superstep, send and receive a number of messages, and handle various remote read and write requests.

Although we have described the BSP computer as an architectural model, one can also view bulk synchrony as a programming model or, indeed, as a kind of programming methodology. The essence of the BSP approach to parallel programming is the notion of the superstep, in which communication and synchronisation are completely decoupled. A "BSP program" is simply one which proceeds in phases, with the necessary global communications taking place between the phases. This approach to parallel programming is applicable to all kinds of parallel architecture: distributed memory architectures, shared memory multiprocessors, and networks of workstations. It provides a consistent, and very general, framework within which to develop portable parallel software for scalable parallel architectures.

Since communication and synchronisation are decoupled in a BSP program, the programmer does not have to worry about problems such as deadlock, which can occur with synchronous message passing. Debugging a BSP program is also made much easier by this decoupling. The barrier at the end of a superstep provides an appropriate breakpoint at which the global state of the parallel computation is well defined and can be interrogated. Debugging and reasoning about the correctness of a BSP program are, therefore, not much more difficult than for a sequential program.

3.2 Cost Modelling

If we define a time step to be the time required for a single local operation, i.e. a basic operation (such as addition or multiplication) on locally held data values, then the performance of any BSP computer can be characterised by three parameters: p = number of processors; l = number of time steps for barrier synchronisation; g = (total number of local operations performed by all processors in one second)/(total number of words delivered by the communications network in one second, in a situation of continuous traffic). There is also, of course, a fourth parameter s, the number of time steps per second. However, since the other parameters are normalised with respect to that one, it can be ignored in the design of algorithms and programs. The parameter g corresponds to the frequency with which non-local memory accesses can be made; in a machine with a higher value of g one must make non-local memory accesses less frequently. Any parallel computing system can be regarded as a BSP computer, and can be benchmarked accordingly to determine its BSP parameters l and g. The BSP model is therefore not prescriptive in terms of the physical architectures to which it applies. Every general purpose parallel architecture can be viewed by an algorithm designer or programmer as simply a point (p, l, g) in the space of all BSP machines.

The time for a superstep S is determined as follows. Let the work w be the maximum number of local computation steps executed by any processor during S. Let h_s be the maximum number of messages sent by any processor during S, and h_r be the maximum number of messages received by any processor during S. The time for S is then at most $w + g \cdot \max\{h_s, h_r\} + l$ steps. The total time required for a BSP computation is easily obtained by adding the times for each superstep to obtain an expression of the form $W + H \cdot g + S \cdot l$, where W, H, S will typically be functions of n and p. Analysing and predicting the cost of a BSP program is, therefore, no more difficult than analysing and predicting the cost of a sequential program.

4 Networks and Routing Methods

The use of the parameters l and g to characterise the communications performance of the BSP computer contrasts sharply with the way in which communications performance is described for most distributed memory architectures on the market today. A major feature of the BSP model is that it lifts considerations of network performance from the local level to the global level. We are thus no longer particularly interested in whether the network is a 2D array, a butterfly or a hypercube, or whether it is implemented in VLSI or in some optical technology. Our interest is in global parameters of the network, such as l and g, which describe its ability to support data communications in a uniformly efficient manner.

In the design and implementation of a BSP computer, the values of l and g which can be achieved will depend on the capabilities of the available technology and the amount of money that one is willing to spend on the communications network. In asymptotic terms, the values of l and g one might expect for various p processor networks are: ring $[l = O(p), g = O(p)]$, 2D array $[l = O(p^{1/2}), g = O(p^{1/2})]$, butterfly $[l = O(\log p), g = O(\log p)]$, hypercube $[l = O(\log p), g = O(1)]$. These asymptotic estimates are based entirely on the degree and diameter properties of the corresponding graph. In a practical setting, the channel capacities, routing methods used, VLSI implementation etc. would also have a significant impact on the actual values of l and g which could be achieved on a given machine. New optical technologies may offer the prospect of further reductions in the values of l and g which can be achieved, by providing a more efficient means of non-local communication than is possible with VLSI.

If we are interested in the problem of designing improved networks and routing methods which reduce g then perhaps the most obvious approach is to concentrate instead on the alternative problem of reducing l. This strategy is suggested by the following simple reasoning: If messages are in the network for a shorter period of time then, given that the network capacity is fixed, it will be possible to insert messages into the network more frequently. In studying BSP algorithms we see that, in many cases, the performance of a BSP computation is limited much more by g than by l. This suggests that in future, when designing networks and routing methods it may be advantageous to accept a significant increase in l in order to secure even a modest decrease in g. This raises a number of interesting architectural questions which have not yet been fully explored. It is also interesting to note that the characteristics of many modern communication systems (slow switching, very high bandwidth) may be very compatible with this alternative approach.

5 BSP Algorithms

In designing an efficient BSP algorithm or program for a problem which can be solved sequentially in time $T(n)$ our goal will, in general, be to produce an algorithm requiring total time $W + H \cdot g + S \cdot l$ where $W(n,p) = T(n)/p$, $H(n,p)$ and $S(n,p)$ are as small as possible, and the range of values for p is as large as possible. In many cases, this will require that we carefully arrange the data distribution so as to minimise the frequency of remote memory references. Another property of interest in BSP algorithm design is the space (or memory) efficiency of the computation. We will use $M(n,p)$ to denote the maximum number of values which any one processor has to store at any point during the computation. In this section we describe some BSP algorithms for matrix multiplication and discuss their time and space complexity.

Many static computations can be conveniently modelled by directed acyclic graphs, where each node corresponds to some simple operation, and the arcs correspond to inputs and outputs. Let C_n denote the directed acyclic graph which has n^3 nodes $v_{i,j,k}$, $0 \leq i,j,k < n$, and arcs from $v_{i,j,k}$ to $v_{i+1,j,k}$, $v_{i,j+1,k}$ and $v_{i,j,k+1}$ where those nodes exist. The graph C_n can be scheduled for a p processor BSP computer by partitioning C_n into $p^{3/2}$ subgraphs, each of which is isomorphic to $C_{n/p^{1/2}}$. Let $s = n/p^{1/2}$ and $C^{\hat{\imath},\hat{\jmath},\hat{k}}$, $0 \leq \hat{\imath},\hat{\jmath},\hat{k} < p^{1/2}$, denote the subset of s^3 nodes $v_{i,j,k}$ in C_n where $i \; div \; s = \hat{\imath}$, $j \; div \; s = \hat{\jmath}$ and $k \; div \; s = \hat{k}$. The following simple schedule for C_n requires $3p^{1/2} - 2$ supersteps: During superstep $s(t)$, each $C^{\hat{\imath},\hat{\jmath},\hat{k}}$ for which $\hat{\imath} + \hat{\jmath} + \hat{k} = t$ is computed by one of the p processors, with no two of them computed by the same processor. ¿From the structure of C_n it is clear that during a superstep, each processor will receive n^2/p values, send n^2/p values, and perform $n^3/p^{3/2}$ computation steps. The total time required for the BSP implementation of any computation which can be modelled by C_n

is therefore at most $n^3/p + (n^2/p^{1/2}) \cdot g + p^{1/2} \cdot l$. [Throughout this paper we will omit the various small constant factors in such formulae.]

Consider the problem of multiplying two $n \times n$ dense matrices A, B to produce C. In [8] it is shown that the product C can be computed using the following set of definitions: For all $0 < i, j, k \leq n$,

$$\begin{aligned} a_{i,j,k} &= a_{i,j-1,k} \\ b_{i,j,k} &= b_{i-1,j,k} \\ c_{i,j,k} &= c_{i,j,k-1} + (a_{i,j,k} \cdot b_{i,j,k}) \end{aligned}$$

where $a_{i,0,k} = a_{i,k}$, $b_{0,j,k} = b_{k,j}$ and $c_{i,j,0} = 0$. These definitions can be directly translated into a labelled version of the directed acyclic graph C_n. The BSP time complexity of matrix multiplication is therefore at most $n^3/p + (n^2/p^{1/2}) \cdot g + p^{1/2} \cdot l$. For the standard n^3 sequential matrix multiplication algorithm, this BSP schedule is optimal in terms of its computation cost $W(n,p) = n^3/p$. It is also optimal in terms of its space complexity. The matrices A and B can be uniformly distributed across the p processors, with each one holding an $n/p^{1/2} \times n/p^{1/2}$ block of the matrix. Given this uniform input distribution, we can schedule the reuse of memory locations in a straightforward way to ensure that no processor will be required to store more than n^2/p values in any superstep. Therefore we have $M(n,p) = n^2/p$.

In [9] a more efficient BSP realisation of the standard n^3 algorithm, due to McColl and Valiant, was described. Its BSP time complexity is $n^3/p + (n^2/p^{2/3}) \cdot g + l$. As in the previous schedule we begin with A, B distributed uniformly but arbitrarily across the p processors. At the end of the computation, the n^2 elements of C should also be distributed uniformly across the p processors. Let $s = n/p^{1/3}$ and $A[\hat{\imath}, \hat{\jmath}]$ denote the $s \times s$ submatrix of A consisting of the elements $a_{i,j}$ where $i \; div \; s = \hat{\imath}$ and $j \; div \; s = \hat{\jmath}$. Define $B[\hat{\imath}, \hat{\jmath}]$ and $C[\hat{\imath}, \hat{\jmath}]$ similarly. Then we have $C[\hat{\imath}, \hat{\jmath}] = \sum_{0 \leq \hat{k} < p^{1/3}} A[\hat{\imath}, \hat{k}] \cdot B[\hat{k}, \hat{\jmath}]$. Let $PROC(\hat{\imath}, \hat{\jmath}, \hat{k})$, $0 \leq \hat{\imath}, \hat{\jmath}, \hat{k} < p^{1/3}$, denote the p processors. In the first superstep each processor $PROC(\hat{\imath}, \hat{\jmath}, \hat{k})$ gets the set of elements in $A[\hat{\imath}, \hat{k}]$ and those in $B[\hat{k}, \hat{\jmath}]$. The cost of this step is $(n^2/p^{2/3}) \cdot g + l$. In the second superstep $PROC(\hat{\imath}, \hat{\jmath}, \hat{k})$ computes $A[\hat{\imath}, \hat{k}] \cdot B[\hat{k}, \hat{\jmath}]$ and sends each one of the $n^2/p^{2/3}$ resulting values to the unique processor which is responsible for computing the corresponding value in C. The cost of this step is $n^3/p + (n^2/p^{2/3}) \cdot g + l$. In the final superstep, each processor computes each of its n^2/p elements of C by adding the $p^{1/3}$ values received for that element. The cost of this step is $n^2/p^{2/3} + l$.

An input-output complexity argument can be used to show that for any BSP implementation of the standard n^3 sequential algorithm, if $W(n,p) = n^3/p$ then $H(n,p) \geq n^2/p^{2/3}$. This second BSP schedule for matrix multiplication therefore provides a realisation of the standard n^3 method which simultaneously achieves the optimal values for computation cost $W(n,p)$, communication cost $H(n,p)$ and synchronisation cost $S(n,p)$. The memory requirement of this algorithm is, however, inferior to the first algorithm. Its memory complexity $M(n,p)$ is $n^2/p^{2/3}$.

6 BSP Programming

In this section we briefly describe the main characteristics of BSP programming. We also compare the BSP approach with two other approaches to parallel programming - data parallelism and message passing.

As was noted earlier, the essence of the BSP approach to parallel programming is the notion of the superstep, in which communication and synchronisation are completely decoupled. A "BSP program" is simply one which proceeds in phases, with the necessary global communications taking place between the phases. One simple way of specifying the data communications in a BSP program is to use remote memory access primitives. The operation *put* deposits locally held data into a remote memory area on another process. The *get* operation reaches into the local memory of another process to copy values held there into a data structure in its own local memory. The put and get operations are both one-sided communication primitives. They do not require the active participation of the other process. In accordance with BSP superstep semantics, they are also both non-blocking. All put and get operations initiated during a superstep will be completed before the start of the next superstep.

Bulk synchronous remote memory access is a very convenient style of programming for BSP computations which can be statically analysed in a straightforward way. It is less convenient for computations where the volumes of data being communicated between supersteps is irregular and data dependent, and where the computation to be performed in a superstep depends on the quantity and form of data received at the start of that superstep. A more appropriate style of programming in such cases is bulk synchronous message passing. In bulk synchronous message passing, a non-blocking *send* operation is used to transfer values held locally into a buffer on the destination process. The values are guaranteed to be in the remote buffer before the start of the next superstep, and can be safely inspected and manipulated by the receiving process at that time.

6.1 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. The BSP approach, as outlined in this paper, aims to offer a more flexible and general style of programming than is provided by data parallelism. The two approaches are not, however, incompatible in any fundamental way. For some applications, the increased 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 extremely important role in terms of providing an analytic framework for performance prediction of the data parallel program.

6.2 Message Passing

Since the early 1980s, message passing has been the dominant programming approach in the area of parallel computing. In recent years, the PVM message passing library [3] 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 which 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 [6] 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 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, scan etc. These standard communication patterns also arise frequently in the design of BSP algorithms. It is important that such structured patterns can be conveniently expressed and efficiently implemented in any BSP programming language, in addition to the more primitive operations such as put and get which generate arbitrary and unstructured communication patterns.

Comparing it to PVM and MPI, it might be argued that the BSP approach offers (a) a simple programming discipline (based on supersteps) which 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.

7 BSP Programming Libraries

The Cray T3D SHMEM library provides primitives for direct remote memory access which can be used for BSP programming. The Oxford BSP Library [10] and the Oxford BSP Toolset [7] both provide a similar set of programming primitives for bulk synchronous remote memory access. The Green BSP Library

[4] provides a set of bulk synchronous message passing primitives based on fixed sized packets. Considerable experience of BSP programming has been gained through the use of these libraries. A number of major projects in universities and in industry are now using them to develop parallel applications, see e.g. [7, 11]. The experience gained in these practical projects would appear to confirm the various claims made above, regarding BSP and its advantages over message passing.

In December 1995, the inaugural meeting of BSP Worldwide was held in Oxford. BSP Worldwide is a new global organisation to coordinate research and development activities in the area of BSP computing, and to work on the standardisation of programming tools for the growing number of software developers who are now adopting this approach. It has recently launched an initiative to produce a standard low level BSP programming library for use with sequential languages such as Fortran and C. An initial proposal for this library is given in [5]. Its main characteristics are as follows:

- Single Program Multiple Data (SPMD) parallelism.
- Primitives for buffered and unbuffered bulk synchronous remote memory access.
- Primitives for buffered bulk synchronous message passing with tagged messages.
- Primitives for address registration to (a) support data communications into static, stack and heap allocated data structures, and (b) support BSP programming in heterogeneous environments.

A number of features which are semantically well defined, such as nested parallelism and subset synchronisation, have been excluded from the initial version of the library since they can have an adverse effect on the predictability of performance.

8 BSP and other models

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

The BSP model provides software developers with an attractive escape route from the world of architecture dependent parallel software. The emergence of the model has also, as we have seen, coincided with the convergence of commercial parallel machine designs to a standard architectural form which is very compatible with the model. These developments have been enthusiastically welcomed by a rapidly growing community of software engineers charged with the task of producing 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". This has led to a new proliferation, this time of models, in the 1990s.

Over the last few years a large number of variants of the BSP model, and alternatives to the BSP model, have been proposed for consideration. The number of such models probably greatly exceeds the number of different architectures that the parallel programmer had to contend with ten years ago! Most of the variants and alternatives have been developed in response to one or both of the following perceptions:

- Barrier synchronisation is an inflexible mechanism for structuring parallel programs.
- Some network characteristics other than l and g have to be taken into account in designing an efficient parallel program.

The only one of these alternative models which has generated any serious interest is the LogP model [2]. LogP differs from BSP in two ways:

- It uses a form of message passing based on pairwise synchronisation.
- It adds a third parameter representing the overhead involved in sending a message.

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.
- Major improvements in network hardware and in communications software have greatly reduced the overhead associated with sending messages.

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 can enable 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 [1] 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.

It is an interesting and important activity to look for alternative models of parallel computation which improve upon what we already have. In encouraging researchers to contribute to our understanding in this area, I would however make the following point. The only sensible way to evaluate an architecture independent model of parallel computation such as BSP, LogP, or the PRAM model, is to consider it in terms of *all* of its properties, i.e. (a) its usefulness as a basis for the design and analysis of algorithms, (b) its universal 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. If we focus on only one of these at a time, then we will simply be replacing the zoo of parallel architectures which we had 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.

References

1. G Bilardi, K T Herley, A Pietracaprina, G Pucci, and P Spirakis. BSP vs LogP. In *Proc. 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1996. (to appear).
2. D 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 *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.
3. A Geist, A Beguelin, J Dongarra, W Jiang, R Manchek, and V Sunderam. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, 1994.
4. M Goudreau, K Lang, S Rao, T Suel, and T Thanasis. Towards efficiency and portability: Programming with the BSP model. In *Proc. 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1996. (to appear).
5. M W Goudreau, J M D Hill, K Lang, W F McColl, S B Rao, D C Stefanescu, T Suel, and T Thanasis. A proposal for the BSP Worldwide Standard Library (preliminary version). Technical report, available via BSP Worldwide home page `http://www.bsp-worldwide.org/`, April 1996.
6. W Gropp, E Lusk, and A Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, MA, 1994.
7. J M D Hill, P I Crumpton, and D A Burgess. The theory, practice, and a tool for BSP performance prediction applied to a CFD application. Technical Report PRG-TR-4-1996, Oxford University Computing Laboratory, 1996. To appear in Proc. Euro-Par '96.
8. W F McColl. Special purpose parallel computing. In A M Gibbons and P Spirakis, editors, *Lectures on Parallel Computation. Proc. 1991 ALCOM Spring School on Parallel Computation*, volume 4 of *Cambridge International Series on Parallel Computation*, pages 261–336. Cambridge University Press, Cambridge, UK, 1993.
9. W F McColl. Scalable computing. In J van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments. LNCS Volume 1000*, pages 46–61. Springer-Verlag, 1995.
10. R Miller. A library for bulk-synchronous parallel programming. In *Proc. British Computer Society Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, December 1993.
11. 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.
12. L G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.