Scalable Computing

W F McColl

Programming Research Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, England

Abstract. Scalable computing will, over the next few years, become the normal form of computing. In this paper we present a unified framework, based on the BSP model, which aims to serve as a foundation for this evolutionary development. A number of important techniques, tools and methodologies for the design of sequential algorithms and programs have been developed over the past few decades. In the transition from sequential to scalable computing we will find that new requirements such as universality and predictable performance will necessitate significant changes of emphasis in these areas. Programs for scalable computing, in addition to being fully portable, will have to be efficiently universal, offering high performance, in a predictable way, on any general purpose parallel architecture. The BSP model provides a discipline for the design of scalable programs of this kind. We outline the approach and discuss some of the issues involved.

1 Introduction

For fifty years, sequential computing has been the normal form of computing. From the early proposal of von Neumann [5] to the present day, sequential computing has grown relentlessly, embracing new technologies as they have emerged and discarding them whenever something better came along. Today it is a huge global industry. The two parts of that industry, hardware and software, are quite different. Most sequential hardware has a life span of a few years. New technologies continually offer new ways of realising the same basic design in a form which is cheaper, more powerful, or both. In contrast, sequential software systems often take many years to develop, and are expected to last for a very long time, preferably for ever.

The key reason for the spectacular success of sequential computing has been the widespread, almost universal, adoption of the basic model proposed by von Neumann. To see why this happened it is necessary to go back further, to the work of Turing [20]. In his theoretical studies, Turing demonstrated that a single general purpose sequential machine could be designed which would be capable of efficiently performing any computation which could be performed by a special purpose sequential machine. Stated more concisely, he demonstrated that

* To appear in LNCS Volume 1000. J. van Leeuwen (Editor). Springer-Verlag (1995).

This work was supported in part by ESPRIT Basic Research Project 9072 - GEPPCOM (Foundations of General Purpose Parallel Computing).

efficient universality was achievable for sequential computations. Around 1944, von Neumann produced his proposal for a general purpose sequential computer which captured the principles of Turing's work in a practical design. The design, which has come to be known as the "von Neumann computer", has served as the basic model for almost all sequential computers produced from the late 1940s to the present time. The stability and universality provided by the von Neumann model has permitted and encouraged the development of high level languages and compilers. These, in turn, have created a large and diverse software industry producing portable applications software which will run with high performance on any sequential computer.

For sequential computing, the von Neumann model has given the two parts of the computing industry what they require to succeed. The hardware industry is provided with a focus for its technological innovation, and the confidence that, since the model is stable, software companies will find that the time and effort involved in developing software can be justified. The software industry meanwhile sees the stability of the model as providing a basis for the high level, cost-effective development of applications software, and also providing a guarantee that the software thus produced will not become obsolete with the next technological change.

Since the earliest days of sequential 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. In this paper we will argue that the evolution to parallel computing can now take place, and will do so over the next few years. We will show that all of the essential requirements for such a transition can now be satisfied. In particular, we will present the BSP model and will argue that it can play a unifying role analogous to that which the von Neumann model has played in the development of sequential computing.

2 Architectures

Parallel computing is not new. For many years, academic research groups and, more recently, companies, have been producing parallel computers and trying to persuade people to write parallel software for them. On most of the early parallel systems, the global communications networks were very poor and scalable performance could only be achieved by designing algorithms and programs which carefully exploited the particular architectural features of the machine. Besides being tedious and time consuming in most cases, this approach typically produced software which could not be easily adapted to run on other machines.

Over the last few years the situation has improved. 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 become 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 global ad-

dress 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. Both message passing and shared memory programming styles will be efficiently supported on these architectures. 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 next Intel microprocessor, currently referred to as the P6, will be the first high volume, commodity microprocessor for the personal computer market with hardware support for multiprocessing. In a couple of years, very low cost multiprocessor personal computers will be available, along with very low cost networking technologies and software allowing these to be assembled into clusters for high performance computing. 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 systems for those who require those computing resources. At present, a good example of a company operating in this end of the market is Cray Research. The CRAY T3D is a very powerful distributed memory architecture based on the DEC Alpha microprocessor. In addition to offering extremely 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 [2]. 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 companies which compete at the top end of the market will, as today, focus their attention not only on highly optimised architectural design, but also on new, and perhaps expensive, technologies which can offer increased performance. For example, some might use optical technologies to achieve more efficient global communications than could be achieved with VLSI systems.

The implementation of a global address space on a distributed memory architecture requires an efficient mechanism for the distributed routing of put and get requests, and of the replies to get requests, through the network of processors. A number of efficient routing and memory management techniques have been

developed for this problem, see e.g. [13, 21, 22]. Consider the problem of packet routing on a p-processor network. Let an *h-relation* denote a routing problem where each processor has at most h packets to send to various processors in the network, and where each processor is also due to receive at most h packets from other processors. Here, a packet is one word of information, such as e.g. a real number or an integer. Using two-phase randomised routing one can, for example, show that every $(\log\ p)$-relation can be realised on a p processor hypercube in $O(\log\ p)$ steps. In [9] a simple and practical randomised method of routing h-relations on an optical communications system is described. The optical system is physically realistic and the method requires only $O(h + \log p \log\log p)$ steps. In [18] a simple and very efficient protocol for routing h-relations using only the total-exchange primitive is described.

The process of architectural convergence which was described above brings with it the hope that we can, over the next few years, establish scalable parallel computing as the normal form of computing, and begin to see the growth of a large and diverse global software industry for scalable computing similar to that which currently exists for sequential computing. The main goal of that industry will be to produce scalable software which will run unchanged, and with high performance, on any architecture, from a cheap multiprocessor PC to a large parallel supercomputer. The next major step towards this goal is to develop a foundation for the architecture independent programming of this emerging range of scalable parallel systems. The BSP approach described in this paper provides just such a foundation. It offers the prospect of achieving both scalable parallel performance and architecture independent parallel software, and provides a framework which permits the performance of parallel and distributed systems to be analysed and predicted in a precise way.

3 The BSP Model

In architectural terms, the BSP model is essentially the standard model described above. A *bulk synchronous parallel (BSP) computer* [13, 21] consists of a set of processor-memory pairs, a global communications network, and a mechanism for the efficient barrier synchronisation of the processors. There are no specialised broadcasting or combining facilities, although these can be efficiently realised in software where required [23]. The model also does not deal directly with issues such as input-output or the use of vector units, although it can be easily extended to do so. 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). [There is also, of course, a fourth parameter, 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 l corresponds to the network latency. The parameter g corresponds to the frequency with which remote memory accesses can be made without affecting the total time for the computation; in a machine with a higher value of g one must make remote memory accesses less frequently. More formally, g is related to the time required to realise h-relations in a situation of continuous message traffic; g is the value such that an h-relation can be performed in $h \cdot g$ time steps. Any scalable parallel 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 scalable 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 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 remote memory accesses 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. As the computational performance of machines continues to grow, we will find that to keep l and g low it will be necessary to continually increase our investment in the communications hardware as a percentage of the total cost of the machine. 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.

For many of the current VLSI-based networks, the values of g and l are very similar. 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 the next section we will 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 explored. The work in [18] contains some interesting initial ideas in this direction. It is also interesting to note that the characteristics of many optical communication systems (slow switching, very high bandwidth) are very compatible with this alternative approach.

A BSP computer operates in the following way. A computation consists of a sequence of S parallel *supersteps* $s(i)$, $0 \leq i < S$, where each superstep consists of a sequence of steps, followed by a barrier synchronisation at which point any remote memory accesses take effect. During a superstep, each processor can perform a number of computation steps on values held locally at the start of the superstep, and send and receive a number of messages corresponding to remote get and put requests. The time for superstep $s(i)$ is determined as follows. Let the work w be the maximum number of local computation steps executed by any processor during $s(i)$, let h_{out} be the maximum number of messages sent by any processor during $s(i)$, and h_{in} be the maximum number of messages received by any processor during $s(i)$. The time for $s(i)$ is then at most $w + \max\{h_{\text{out}}, h_{\text{in}}\} \cdot g + l$ steps. The total time required for a BSP computation is easily obtained by adding the times for each of the S supersteps. This will produce an expression of the form $W + H \cdot g + S \cdot l$ where W, H, S will typically be functions of n and p. 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.

4 BSP Algorithm Design

To illustrate a number of the important issues in BSP algorithm design we will consider several computational problems involving matrices.

4.1 Dense Matrix-Vector Multiplication

In this section we consider the problem of multiplying an $n \times n$ matrix M by an n-element vector v on p processors, where M, v are both dense. In [3], the BSP cost of this problem, for both sparse and dense matrices, was theoretically and experimentally analysed. The BSP algorithm which we now describe was shown there. Its complexity is $n^2/p + (n/p^{1/2}) \cdot g + l$. [Throughout this paper we will omit the various small constant factors in such formulae.]

For dense M and $p \leq n$, the standard n^2 sequential algorithm can be adapted to run on a p processor BSP machine as follows. The matrix elements are initially distributed uniformly across the p processors, with each processor holding an $n/p^{1/2} \times n/p^{1/2}$ submatrix of M. [In [3] this is referred to as a "block-block" distribution.] The vectors u and v are also uniformly distributed across the machine, with n/p elements of u, and n/p elements of v, allocated to each processor. In the first superstep, each processor gets the set of all $n/p^{1/2}$ vector elements v_j for which it holds an $m_{i,j}$. The cost of this step is $(n/p^{1/2}) \cdot g + l$. In the second superstep, each processor k computes $u_i^k = \sum_{a \leq j < a+n/p^{1/2}} m_{i,j} \cdot v_j$ for each of the $n/p^{1/2}$ subrows of M which it holds, and sends the partial sum u_i^k to the processor which holds u_i. The time required for this step is $n^2/p + (n/p^{1/2}) \cdot g + l$. In the third and final superstep, each processor computes the value of each of its vector elements u_i by adding the $p^{1/2}$ values u_i^k which it receives. The time required for this final step is $n/p^{1/2} + l$.

An input-output complexity argument, similar to those in [1, 11], can be used to show that for any BSP algorithm which computes $u = M \cdot v$, if $W(n,p) = n^2/p$ then $H(n,p) \geq n/p^{1/2}$. Noting that the n^2 sequential computation cost is itself optimal we see that the above method is in a strong sense a best possible BSP algorithm for this problem. It simultaneously achieves the optimal computation cost $W(n,p) = n^2/p$, the optimal communication cost $H(n,p) \cdot g = (n/p^{1/2}) \cdot g$ and the optimal synchronisation cost $S(n,p) \cdot l = l$. Other matrix distributions, such as the "block-grid" distribution [3], also give these bounds. [The block-grid distribution is defined as follows. Let *PROC(r,c)*, $0 \leq r, c < p^{1/2}$, denote the p processors. The matrix elements $m_{i,j}$, $0 \leq i, j < n$, are uniformly distributed across the processors, with $m_{i,j}$ allocated to *PROC(*i **div** *(*$n/p^{1/2}$*),* j **mod** $p^{1/2}$*).*]

4.2 Matrix Multiplication

We now consider the problem of multiplying two $n \times n$ dense matrices A, B on p processors. For $p \leq n^2$, the standard n^3 sequential algorithm can be adapted to run on a p processor BSP machine as follows. Each processor computes an $(n/p^{1/2}) \times (n/p^{1/2})$ submatrix of $C = A \cdot B$. To do so it will require $n^2/p^{1/2}$ elements from A and the same number from B. If A and B are both distributed uniformly across the p processors, with each processor holding n^2/p of the elements from each matrix, then the total time required for this algorithm will be $n^3/p + (n^2/p^{1/2}) \cdot g + l$.

We now describe a more efficient BSP implementation of the standard n^3 algorithm, due to the author and L. G. Valiant. Its BSP complexity is $n^3/p + (n^2/p^{2/3}) \cdot g + l$. As in the previous algorithm 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 $A[i,j]$ denote the $s \times s$ submatrix of A consisting of the elements $a_{\hat{\imath},\hat{\jmath}}$ where $\hat{\imath}$ **div** $s = i$ and $\hat{\jmath}$ **div** $s = j$. Define $B[i,j]$ and $C[i,j]$ similarly. Then we have $C[i,j] = \sum_{0 \leq k < p^{1/3}} A[i,k] \cdot B[k,j]$. Let *PROC(i,j,k)*, $0 \leq i, j, k < p^{1/3}$, denote the p processors.

In the first superstep each processor *PROC(i,j,k)* gets the set of elements in $A[i,j]$ and those in $B[j,k]$. The cost of this step is $(n^2/p^{2/3}) \cdot g + l$. In the second superstep *PROC(i,j,k)* computes $A[i,j] \cdot B[j,k]$ 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 also be used to show that for any BSP implementation of the standard n^3 sequential algorithm, if $W(n,p) = n^3/p$ then $H(n,p) \geq n^2/p^{2/3}$. The above algorithm is therefore a best possible BSP implementation of the standard n^3 method in the sense that it simultaneously achieves the optimal values for $W(n,p)$, $H(n,p)$ and $S(n,p)$.

4.3 LU Decomposition

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. In [14] it is shown that the LU decomposition of an $n \times n$ non-singular matrix A can be computed (without pivoting) using the following set of definitions: For all $0 \leq k \leq i,j < n$,

$$
\begin{aligned}
u_{k,k,k} &= a_{k,k,k-1} \\
l_{i,j,k} &= a_{i,j,k-1}/u_{i,j,k} \text{ if } j = k, \text{ and } l_{i,j-1,k} \text{ otherwise.} \\
u_{i,j,k} &= a_{i,j,k-1} \text{ if } i = k, \text{ and } u_{i-1,j,k} \text{ otherwise.} \\
a_{i,j,k} &= a_{i,j,k-1} - (l_{i,j,k} \cdot u_{i,j,k})
\end{aligned}
$$

where $a_{i,j,-1} = a_{i,j}$. These definitions can be directly translated into a directed acyclic graph which is a subgraph of C_n. Therefore, to produce a BSP algorithm for LU decomposition it is sufficient to schedule C_n for a p processor BSP computer. We can do this 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, such as LU decomposition, is therefore at most $n^3/p + (n^2/p^{1/2}) \cdot g + p^{1/2} \cdot l$.

4.4 Solution of a Triangular Linear System

Let A be an $n \times n$ non-singular, lower triangular matrix, and b be an n-element vector. The linear system $A \cdot x = b$ can be solved by back substitution using the recurrence $x_i = (b_i - \sum_{1 \leq j < i} a_{i,j} \cdot x_j)/a_{i,i}$ for $0 \leq i < n$. As in the case of LU decomposition, we can produce an efficient BSP implementation of back substitution by realising the computation as a directed acyclic graph and then scheduling the graph. Let D_n denote the directed acyclic graph which has n^2 nodes $v_{i,j}$, $0 \leq i, j < n$, and arcs from $v_{i,j}$ to $v_{i+1,j}$ and $v_{i,j+1}$ where those nodes exist. The back substitution recurrence above can be reformulated into the following set of definitions: For all $0 \leq j \leq i < n$,

$$\begin{aligned} t_{i,i} &= (b_i - t_{i,i-1})/a_{i,i} \\ t_{i,j} &= a_{i,j} \cdot r_{i,j} + t_{i,j-1} \text{ if } i > j \\ r_{i,i} &= t_{i,i} \\ r_{i,j} &= r_{i-1,j} \text{ if } i > j \\ x_i &= t_{i,i} \end{aligned}$$

where $t_{i,-1} = 0$. These definitions can be directly translated into a directed acyclic graph which is a subgraph of D_n. The graph D_n can be scheduled for a p processor BSP computer in a very similar manner to that used for C_n. In this case, we partition D_n into p^2 subgraphs, each of which is isomorphic to $D_{n/p}$.

Let $s = n/p$ and $C^{\hat{\imath},\hat{\jmath}}$, $0 \leq \hat{\imath}, \hat{\jmath} < p$, denote the set of nodes $v_{i,j}$ in D_n where i **div** $s = \hat{\imath}$ and j **div** $s = \hat{\jmath}$. The following simple schedule for D_n requires $2p-1$ supersteps: During superstep $s(t)$, each $C^{\hat{\imath},\hat{\jmath}}$ for which $\hat{\imath} + \hat{\jmath} = t$ is computed by one of the p processors, with no two of them computed by the same processor. From the structure of D_n it is clear that during a superstep, each processor will receive n/p values, send n/p values, and perform n^2/p^2 computation steps. The total time required for the BSP implementation of any computation which can be modelled by D_n, such as back substitution, is therefore at most $n^2/p + n \cdot g + p \cdot l$.

In some situations, using more processors in a BSP architecture will actually *increase* the runtime. For example, consider a BSP architecture based on a ring, with $l = g = p$. The runtime of our BSP algorithm for the solution of an $n \times n$ triangular linear system will be $n^2/p + np + p^2$. This runtime is minimised when $p = n^{1/2}$. Increasing the number of processors beyond this value will increase the runtime.

4.5 Sparse Matrix-Vector Multiplication

We now return to the problem of computing $u = M \cdot v$. Our interest now is in the case where the matrix M is sparse. Sparse matrix-vector multiplication is a problem of fundamental importance in scientific computing. It is at the heart of many supercomputing applications which use iterative methods to solve very large linear systems. To enable the multiplication to be performed repeatedly with no additional data redistribution we require that the result vector $u = M \cdot v$ should be distributed across the parallel machine in the same way as the input

vector v, i.e. the processor holding v_i at the start of the computation should hold u_i at the end of the computation.

Let $C(r,d)$ denote the adjacency matrix of the directed r-ary, d-dimensional hypercube graph. The nodes of this graph form a d-dimensional grid of $n = r^d$ points which are numbered lexicographically. Each node has directed arcs to itself and to its immediate neighbours in each dimension. For the purposes of discussion we will consider just four $n \times n$ sparse matrices. Three of the four are instances of $C(r,d)$. They are: 2D-MESH $= C(n^{1/2}, 2)$, 3D-MESH $= C(n^{1/3}, 3)$ and HYPERCUBE $= C(2, \log n)$. Matrices of this kind are often used to model finite-difference operators in the solution of partial differential equations [3]. The fourth matrix has a random structure in which each row and each column contains four nonzeros. We will refer to it as the EXPANDER matrix. [Note. The value four is not particularly significant. We could have chosen any small integer value greater than one.]

The theoretical and experimental analysis in [3] shows that, compared with random data distributions, matrix-based distributions such as block-grid can offer some reductions in the BSP communication cost $H(n,p)$ of most sparse matrix-vector multiplication problems. It also shows that in many cases, much more significant reductions in $H(n,p)$ can be achieved by using a data distribution based on an efficient decomposition of the underlying graph. For example, the nodes of the 2D-MESH graph can be partitioned into p regions, each of which corresponds to a 2D-MESH on n/p nodes. The corresponding data distribution for matrix elements gives a BSP algorithm for $u = M \cdot v$, where M is the 2D-MESH matrix, with total cost $n/p + (n^{1/2}/p^{1/2}) \cdot g + l$. The same approach, applied to the 3D-MESH matrix, gives a BSP algorithm with total cost $n/p + (n^{2/3}/p^{2/3}) \cdot g + l$ and, applied to the HYPERCUBE matrix, gives a BSP algorithm with total cost $(n \log n)/p + ((n \log p)/p) \cdot g + l$. In each case we minimise the communication cost of the algorithm by partitioning the nodes of the graph into p equal sized subsets in a way which minimises the number of arcs between different subsets. Lower bounds on the efficiency of such partitions can be derived from known isoperimetric inequalities in graph theory, see e.g. [4].

For the EXPANDER matrix, the best upper bound which we have is the trivial one, $n/p + (n/p) \cdot g + l$, which can be obtained by randomly distributing the matrix elements. Techniques similar to those in [12] can be used to show that for the EXPANDER matrix there is no partition of the nodes into p equal sized subsets which gives a value for $H(n,p)$ which is less than the trivial n/p. Therefore, for the EXPANDER matrix, $u = M \cdot v$ is an inherently non-local problem.

5 BSP Programming

Although we have described the BSP computer as an architectural model, one can also view bulk synchrony as a programming discipline. 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 software for scalable computing.

Since the early 1980s, message passing based on synchronised point-to-point communication has been the dominant programming approach in the area of parallel computing. What are the advantages of BSP programming over message passing? On heterogeneous parallel architectures, in which the individual nodes are quite varied, the answer is probably that there is not a lot to be gained from the BSP approach. On homogeneous distributed memory architectures with low capacity global communications, the two approaches will be broadly similar in terms of the efficiency which can be achieved. On shared memory architectures and on modern distributed memory architectures with powerful global communications, synchronised message passing is likely to be less efficient than BSP programming, where communication and synchronisation are decoupled. This will be especially true on those modern distributed memory architectures which have hardware support for non-blocking direct remote memory access (1-sided communications). 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. Comparing it to message passing, the BSP approach offers (a) a higher level of abstraction for the programmer, (b) a cost model for performance analysis and prediction which is simpler and compositional, and (c) more efficient implementations on many machines.

Some message passing systems provide primitives for various specialised communication patterns which arise frequently in message passing programs. These include broadcast, scatter, gather, complete 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. The efficient implementation of broadcasting and combining on a BSP architecture is discussed in [23].

Data parallelism is an important niche within the field of scalable 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. [19]. 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 scheduling 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.

In Figure 1 we give a pseudocode version of the BSP algorithm for dense matrix-vector multiplication, where the data distribution is defined to be block-grid. For simplicity we assume that n is a multiple of p, and that p is a perfect square. The assumption of such an ideal match between problem size and machine size is, of course, unrealistic in practice. It does however permit us to give a clear and concise description of the main points of the BSP algorithm and its implementation, without having to give all of the technical details required for the general case. The various non-standard constructs used in the pseudocode are informally described in Figure 2.

Given integers *ndivp, sqrtp.*

```
const p = sqrtp^2;
const n = ndivp*p;
const b = n/sqrtp;
const side = 0 .. n-1;
var amem : array [side,side] of real with blocksize [b,b];
view A[i,j] = amem[i,b*(j mod sqrtp)+(j div sqrtp)];
var vmem : array [side,(0 .. sqrtp)] of real with blocksize [ndivp,sqrtp+1];
view v[i] = vmem[i,0];
view vsums[i,j] = vmem[i,j];
par for (i ← 0, b .. n-1 ; j ← 0 .. sqrtp-1)
        at A[i,j];
        const jgroup = j, j+sqrtp .. n-1;
        var t : real;
        get v[k] for k ← jgroup;
        seq for bi ← i .. i+b-1
                t := 0.0;
                seq for bj ← jgroup
                        t := t+A[bi,bj]*v[bj];
                put t in vsums[bi,j+1];
par for k ← 0, ndivp .. n-1
        at v[k];
        seq for w ← k .. k+ndivp-1
                v[w] := 0.0;
                seq for x ← 1 .. sqrtp
                        v[w] := v[w]+vsums[w,x];
```

Fig. 1. *BSP pseudocode for an $n \times n$ matrix vector product on p processors.*

var *A :* **array** *[(1 .. c*r),(1 .. d*s)]* **of real with blocksize** *[r,s];*

Declares a two-dimensional array A of size $c \cdot r \times d \cdot s$, in which the elements of A are to be partitioned into $c \cdot d$ contiguous blocks, each of size $r \times s$. The storage for each complete block will be allocated on a single processor-memory pair, i.e. a block will not be split across two or more memory modules. The arrangement of the various complete blocks will be made so as to uniformly distribute them across the set of memory modules in the machine.

get *v[i]* **for** $i \leftarrow 1 \,..\, m$*;*

Get a local copy of the variables $v[1], v[2], \ldots, v[m]$ for use in the operations of the next superstep. If variable $v[i]$ is already held locally then the **get** $v[i]$ operation has no effect. [There is an implicit barrier synchronisation after any sequence of **get** statements.] Within a superstep, any attempt to access a value which is not held locally will result in a run-time error.

put *u* **in** *v;*

Put the value of the local variable u in the remote variable v. [This remote write will be completed by the end of the current superstep.]

at *v;*

Execute this parallel thread on the processor-memory pair to which the variable v has been allocated. The inclusion of an **at** declaration in a thread is optional.

Fig. 2. Notations used in the BSP pseudocode.

6 BSP Programming Languages

We noted above that the BSP model was not prescriptive in terms of the physical architectures to which it applies. It is also not prescriptive in terms of the programming languages and programming styles to which it applies. In this section we briefly describe a number of the programming languages and libraries which are currently available to support BSP programming, and some which are currently under development.

The PVM message passing library [8] is widely implemented and widely used. The MPI message passing interface [16] is more elaborate. It supports blocking and non-blocking point-to-point communication and a number of collective communications (broadcast, scatter, gather, reduction etc.). Although neither of these libraries is directly aimed at supporting BSP programming, they can be used for that purpose.

The Oxford BSP Library [17] consists of a set of subroutines which can be called from standard sequential languages such as Fortran and C. The core of

the Library consists of just six routines: `bsp_start`, `bsp_finish`, `bsp_sstep`, `bsp_sstep_end`, `bsp_fetch` and `bsp_store`. The first two are for process management, the next two are for barrier synchronisation, and the last two are for communication. Higher level operations such as broadcast and reduction are also available. The Library supports a static SPMD style of BSP programming. It has been implemented on a large number of machines and is being used by a growing community of applications developers to produce source codes which run unchanged and efficiently on a wide variety of parallel and distributed systems. Generic versions of the Library are freely available by ftp to run on any homogeneous parallel UNIX machine with at least one of: PVM, PARMACS, TCP/IP, or System V Shared Memory primitives. Highly optimised native implementations have been produced for the IBM SP1/SP2, the SGI Power Challenge, the CRAY T3D and other machines. Some preliminary benchmarking studies have also been carried out for these systems, to estimate the values of l and g which a programmer using the native implementation should expect.

GPL is a new programming language for scalable computing. It is being developed at Oxford by the author and Quentin Miller as part of ESPRIT Project 9072 - GEPPCOM (Foundations of General Purpose Parallel Computing). The first prototype compiler is currently operational. A preliminary description of some of the ideas behind this work can be found in [15]. The language is designed to permit the efficient, high level programming of static and dynamic BSP computations, and to permit the performance of those programs to be accurately analysed and predicted. The notations used in the pseudocode example above are similar in style to those of GPL. The language is procedural, explicitly parallel and strongly-typed. It allows the programmer to explicitly control scheduling, synchronisation, memory management, combining and other properties of a program which may be crucial to achieving efficient, scalable and predictable performance while retaining portability. For example, if run on a BSP architecture, the program will have access to constants `G`, `P` and `L` corresponding to the BSP parameters of the machine. The program can use these to optimise the computation. In a multiuser system, the constants might be supplied by the operating system at run time. [At some point in the future of BSP computing it may also become customary for applications programs to assist in optimising resource allocation on multiuser multiprocessors by indicating to the operating system the p, g and l resources that it could efficiently utilise (or cope with!). For example, the dense matrix-vector multiplication code in the previous section might contain a declaration indicating $p_{\max} = n$, $g_{\max} = n/p^{1/2}$ and $l_{\max} = n^2/p$ in some way.] As with any language for high performance programming, a key requirement of the GPL language and its implementations is that there should be an accurate cost model which will be applicable to any implementation of the language. With a reliable cost model of this kind, the programmer will be able to make appropriate design decisions to achieve the highest possible performance.

Split-C [7] is a parallel extension of C which supports efficient access to a global address space on current distributed memory architectures. Like GPL, it aims to support careful engineering and optimisation of portable parallel pro-

grams by providing a cost model for performance prediction. The language extension is based on a small set of global access primitives and simple memory management declarations which support both bulk synchronous and message driven styles of parallel programming.

BSP-L [6] is an experimental BSP programming language under development at Harvard. The language is being used to explore the effectiveness of various constructs which might be added to conventional languages such as Fortran and C to support BSP programming. An important objective of this work is to reach a much better understanding of the issues involved in designing optimising compilers and efficient run time systems for programming languages based on the BSP model.

7 Challenges

The study of BSP program design has only recently begun. In contrast to sequential computing, we do not yet have well developed techniques, tools and methodologies for dealing with the specification, refinement, transformation, verification, modularity, reusability, fault tolerance, and other important aspects of BSP programs. Nor do we have a blueprint for the development of a high performance operating system for multiuser BSP computers. These, and many other problems, remain challenges for the future of scalable computing.

References

1. A Aggarwal, A K Chandra, and M Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71:3–28, 1990.
2. R H Arpaci, D E Culler, A Krishnamurthy, S G Steinberg, and K Yelick. Empirical evaluation of the CRAY-T3D: A compiler perspective. In *Proc. 22nd Annual International Symposium on Computer Architecture*, June 1995.
3. R H Bisseling and W F McColl. Scientific computing on bulk synchronous parallel architectures. Technical Report 836, Department of Mathematics, University of Utrecht, December 1993. Short version appears in Proc. 13th IFIP World Computer Congress. Volume I (1994), B. Pehrson and I. Simon, Eds., Elsevier, pp. 509-514.
4. B Bollobás. *Random Graphs.* Academic Press, 1985.
5. A W Burks, H H Goldstine, and J von Neumann. *Preliminary discussion of the logical design of an electronic computing instrument. Part 1, Volume 1.* The Institute of Advanced Study, Princeton, 1946. Report to the U.S. Army Ordnance Department. First edition, 28 June 1946. Second edition, 2 September 1947. Also appears in *Papers of John von Neumann on Computing and Computer Theory*, W Aspray and A Burks, editors. Volume 12 in the Charles Babbage Institute Reprint Series for the History of Computing, MIT Press, 1987, 97-142.
6. T Cheatham, A Fahmy, D C Stefanescu, and L G Valiant. Bulk synchronous parallel computing - a paradigm for transportable software. In *Proc. 28th Hawaii International Conference on System Science*, January 1995.

7. D E Culler, A Dusseau, S C Goldstein, A Krishnamurthy, S Lumetta, T von Eicken, and K Yelick. Parallel programming in Split-C. In *Proc. Supercomputing '93*, pages 262–273, November 1993.
8. 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.
9. M Gereb-Graus and T Tsantilas. Efficient optical communication in parallel computers. In *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 41–48, 1992.
10. A M Gibbons and P Spirakis, editors. *Lectures on Parallel Computation*, volume 4 of *Cambridge International Series on Parallel Computation*. Cambridge University Press, Cambridge, UK, 1993.
11. J W Hong and H T Kung. I/O complexity: The red-blue pebble game. In *Proc. 13th Annual ACM Symposium on Theory of Computing*, pages 326–333, 1981.
12. G Manzini. Sparse matrix vector multiplication on distributed architectures: Lower bounds and average complexity results. *Information Processing Letters*, 50(5):231–238, June 1994.
13. W F McColl. General purpose parallel computing. In Gibbons and Spirakis [10], pages 337–391.
14. W F McColl. Special purpose parallel computing. In Gibbons and Spirakis [10], pages 261–336.
15. W F McColl. BSP programming. In G E Blelloch, K M Chandy, and S Jagannathan, editors, *Specification of Parallel Algorithms. Proc. DIMACS Workshop, Princeton, May 9-11, 1994*, volume 18 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 21–35. American Mathematical Society, 1994.
16. Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, May 1994.
17. 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. A revised and extended version of this paper is available by anonymous ftp from ftp.comlab.ox.ac.uk in directory /pub/Packages/BSP along with the Oxford BSP Library software distribution.
18. S Rao, T Suel, T Tsantilas, and M Goudreau. Efficient communication using total-exchange. In *Proc. 9th International Parallel Processing Symposium*, 1995.
19. D Skillicorn. *Foundations of Parallel Programming*, volume 6 of *Cambridge International Series on Parallel Computation*. Cambridge University Press, Cambridge, UK, 1994.
20. A M Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society. Series 2*, 42:230–265, 1936. Corrections, *ibid.*, 43 (1937), 544-546.
21. L G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
22. L G Valiant. General purpose parallel architectures. In J van Leeuwen, editor, *Handbook of Theoretical Computer Science : Volume A, Algorithms and Complexity*, pages 943–971. North Holland, 1990.
23. L G Valiant. A combining mechanism for parallel computers. In F Meyer auf der Heide, B Monien, and A L Rosenberg, editors, *Parallel Architectures and Their Efficient Use. Proceedings of the First Heinz Nixdorf Symposium, Paderborn, November 1992. LNCS Vol. 678*, pages 1–10. Springer-Verlag, 1993.