A Bridging Model for Parallel Computation

The success of the von Neumann model of sequential computation is attributable to the fact that it is an efficient bridge between software and hardware: high-level languages can be efficiently compiled on to this model; yet it can be efficiently implemented in hardware. The author argues that an analogous bridge between software and hardware is required for parallel computation if that is to become as widely used. This article introduces the bulk-synchronous parallel (BSP) model as a candidate for this role, and gives results quantifying its efficiency both in implementing high-level language features and algorithms, as well as in being implemented in hardware.

Leslie G. Valiant

In a conventional sequential computer, processing is channelled through one physical location. In a parallel machine, processing can occur simultaneously at many locations and consequently many more computational operations per second should be achievable. Due to the rapidly decreasing cost of processing, memory, and communication, it has appeared inevitable for at least two decades that parallel machines will eventually displace sequential ones in computationally intensive domains. This, however, has not yet happened. In order to have a chance of rectifying this situation it is necessary to identify what is missing in our understanding of parallel computation that is present in the sequential case, making possible a huge and diverse industry.

We take the view that the enabling ingredient in sequential computation is a central unifying model, namely the von Neumann computer. Even with rapidly changing technology and architectural ideas, hardware designers can still share the common goal of realizing efficient von Neumann machines, without having to be too concerned about the software that is going to be executed. Similarly, the software industry in all its diversity can aim to write programs that can be executed efficiently on this model, without explicit consideration of the hardware. Thus, the von Neumann model is the connecting bridge that enables programs from the diverse and chaotic world of software to run efficiently on machines from the diverse and chaotic world of hardware.

Our claim is that what is required before general purpose parallel computation can succeed is the adoption of an analogous unifying *bridging model* for parallel computation. A major purpose of such a model is simply to act as a standard on which people can agree. In order to succeed in this role, however, the model has to satisfy some stringent quantitative requirements, exactly as does the von Neumann model. Despite the clear benefits that might flow from the adoption of a bridging model, relatively little effort appears to have been invested in discovering one. Some very relevant issues but in a slightly different context are discussed by Snyder in [23].

In this article we introduce the *bulk-synchronous parallel* (BSP) model and provide evidence that it is a viable candidate for the role of bridging model. It is intended neither as a hardware nor programming model, but something in between. In justifying the BSP for this role, our main argument is that when mapping high-level programs to actual machines in a great variety of contexts, little efficiency is lost if we utilize this single model. The adoption of such a standard can be expected to insulate software and hardware development from one another and make possible both general purpose machines and transportable software.

The quantitative arguments for the model are mainly *efficient universality* results. In three sections, efficient implementations on the model of high-level language features and algorithms are discussed. In two others, implementations of the model in hardware are described. In all cases, we aim to achieve *optimal* simulations, meaning the time taken is optimal to within constant multiplicative factors which are independent of the number of processors and usually small. We wish to avoid logarithmic losses in efficiency. Although we express the results asymptotically, we regard the model as neutral about the number of processors, be it two or one million. This is justified whenever the constants are indeed small.

Since the difficulties of programming present severe potential obstacles to parallel computing, it is important to give the programmer the option to avoid the onerous burdens of managing memory, assigning communication and performing low-level synchronization. A major feature of the BSP model is that it provides this option with optimal efficiency (i.e., within constant factors) provided the programmer writes programs with sufficient *parallel slackness*. This means programs are written for v virtual parallel processors to run on p physical processors where v is rather larger than p (e.g., $v = p \log p$). The slack is exploited by the compiler to schedule and pipeline computation and communication efficiently. The high-level languages that could be compiled in this mode could allow a virtual shared address space. The program would have to be so expressed that v parallel instruction streams could be compiled from it. While a PRAM language [6, 11] would be ideal, other styles also may be appropriate.

We note that in a general-purpose setting some slack may be unavoidable if parallel programs are to be compiled efficiently. Certainly, the prospects for compiling sequential code into parallel code, which is the extreme opposite case of $v=1$, look bleak. The intermediate case of $p=v$ looks unpromising also if we are aiming for optimality. Hence the discipline implied, that of using fewer processors than the degree of parallelism available in the program, appears to be an acceptable general approach to computation-intensive problems. The importance of slack has been emphasized earlier in [12, 27].

It is worth pointing out that while these automatic memory and communication management techniques are available, the model does not make their use obligatory. For the purpose of reducing the amount of slack required, improving constant factors in runtime, or avoiding hashing (as used by the automatic memory management scheme), the programmer may choose to keep control of these tasks. We shall give some illustrative examples of bulk-

*The preparation of this paper was supported in part by the National Science Foundation under grants DCR-86-00379 and CCR-89-02500. A preliminary version appeared as reference [28].

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1990 ACM 0001-0782/90/0800-0103 $1.50

synchronous algorithms that are appropriate for this model.

It is striking that despite the breadth of relevant research in recent years, no substantial impediments to general-purpose parallel computation as we interpret it here have been uncovered. This contrasts with noncomputability and NP-completeness results that explain the intractability of numerous other computational endeavors that had been pursued. Many of the results that have been obtained, and to which we shall refer here in justification of the BSP model, are efficient universality results in the style of Turing's theorem about universal machines [24]. Hence, the BSP model can be viewed as a pragmatic embodiment of these positive results much as the von Neumann model is a pragmatic embodiment of Turing's theorem.

A major purpose of such a model is simply to act as a standard on which people can agree.

The BSP Model

The BSP model of parallel computation or a bulk-synchronous parallel computer (BSPC) is defined as the combination of three attributes:

1. A number of *components,* each performing processing and/or memory functions;
2. A *router* that delivers messages point to point between pairs of components; and
3. Facilities for synchronizing all or a subset of the components at regular intervals of L time units where L is the *periodicity* parameter. A computation consists of a sequence of *supersteps.* In each superstep, each component is allocated a task consisting of some combination of local computation steps, message transmissions and (implicitly) message arrivals from other components. After each period of L time units, a global check is made to determine whether the superstep has been completed by all the components. If it has, the machine proceeds to the next superstep. Otherwise, the next period of L units is allocated to the unfinished superstep.

The definition is chosen to embody the simplest capabilities that suffice for our purposes. In separating the components from the router, we emphasize that the tasks of computation and communication can be separated. The function of the router is to deliver messages point to point. It is specifically intended for implementing storage accesses between distinct components. It assumes no combining, duplicating or broadcasting facilities. Similarly, the synchronization mechanism described captures in a simple way the idea of global synchronization at a controllable level of coarseness. Realizing this in hardware provides an efficient way of implementing tightly synchronized parallel algorithms, among others, without overburdening the programmer. We note that there exist alternative synchronization mechanisms that could have been substituted which achieve the same purpose. For example, the system could continuously check whether the current superstep is completed, and allow it to proceed to the next superstep as soon as completion is detected. Provided a minimum amount of L time units for this check is charged, the results of the run-time analysis will not change by more than small constant factors.

The synchronization mechanism can be switched off for any subset of the components; sequential processes that are independent of the results of processes at other components should not be slowed down unnecessarily. When synchronization is switched off at a processor it can proceed without having to wait for the completion of processes in the router or in other components. Also, operations local to the processor will not automatically slow down computations elsewhere. On the other hand, even when this mechanism is switched off, a processor can still send and receive messages and use this alternative method for synchronization. If performance guarantees are expected of this alternative synchronization mechanism, assumptions have to be made about the router; for example, it might be assumed that each message is delivered within a certain expected amount of time of being sent. In justifying the BSP model, we use the barrier-style synchronization mechanism alone and make no assumptions about the relative delivery times of the messages within a superstep. In the simulations, local operations are carried out only on data locally available before the start of the current superstep.

The value of the periodicity L may be controlled by the program, even at runtime. The choice of its value is constrained in opposite directions by hardware and software considerations. Clearly, the hardware sets lower bounds on how small L can be. The software, on the other hand, sets upper bounds on L since the larger it is, the larger the granularity of parallelism that has to be exhibited by the program. This is because, to achieve optimal processor utilization, in each superstep each processor has to be assigned a task of approximately L steps that can proceed without waiting for results from other processors. We note that along with the tension between these two factors, there is also the phenomenon that a small L, while algorithmically beneficial in general, may not yield any further advantages below a certain value.

In analyzing the performance of a BSP computer, we assume that in one time unit an operation can be computed by a processing component on data available in memory local to it. The basic task of the router is to realize arbitrary *h-relations,* or, in other words, supersteps in which each component sends and is sent at most h messages. We have in mind a charge of $\bar{g}h + s$ time units for realizing such an h-relation. Here $\bar{g}$ defines the basic throughput of the router when in continuous use and s the latency or startup cost. Since our

interest pertains only to optimal simulations, we will always assume that h is large enough that $\bar{g}h$ is at least of comparable magnitude to s. If $\bar{g}h \geq s$, for example, and we let $g = 2\bar{g}$, then we can simply charge gh time units for an h-relation and this will be an overestimate (by a factor of at most two). In this article we shall, therefore, define g to be such that h-relations can be realized in time gh for h larger than some h_0. This g can be regarded as the ratio of the number of local computational operations performed per second by all the processors, to the total number of data words delivered per second by the router. Note that if $L \geq gh_0$ then every h-relation for $h < h_0$ will be charged as an h_0-relation.

Even in a fixed technology we think of the parameter g as being controllable, within limits, in the router design. It can be kept low by using more pipelining or by having wider communication channels. Keeping g low or fixed as the machine size p increases incurs, of course, extra costs. In particular, as the machine scales up, the hardware investment for communication needs to grow faster than that for computation. Our thesis is that if these costs are paid, machines of a new level of efficiency and programmability can be attained.

We note that the von Neumann model as generally understood leaves many design choices open. Implementations incorporating some additions, such as memory hierarchies, do not necessarily become inconsistent with the model. In a similar spirit we have left many options in the BSP computer open. We allow for both single and multiple instruction streams. While it will be convenient in this article to assume that each component consists of a sequential von Neumann processor attached to a block of local memory, we do not exclude other arrangements. Also, we can envisage implementations of the BSP model that incorporate features for communication, computation or synchronization that are clearly additional to the ones in the definition but still do not violate its spirit.

A formalization of perhaps the simplest instance of the BSP model is described in [29] where it is called an XPRAM. A fuller account of the simulation results as well as of their proofs can be found there.

Automatic Memory Management on the BSPC

High-level languages enable the programmer to refer to a memory location used in a program by a symbolic address rather than by the physical address at which it is stored. For sequential machines, conventional compiler techniques are sufficient to generate efficient machine code from the high-level description. In the parallel case, where many accesses are made simultaneously and the memory is distributed over many components, new problems arise. In particular, there is the primary problem of allocating storage in such a way that the computation will not be slowed down by memory accesses being made unevenly and overloading individual units.

The most promising method known for distributing memory accesses about equally in arbitrary programs is hashing. The motivating idea is that if the memory words are distributed among the memory units randomly, independently of the program, then the accesses to the various units should be about equally frequent. Since, however, the mapping from the symbolic addresses to the physical addresses has to be efficiently computable, the description of the mapping has to be small. This necessitates that a pseudo-random mapping or hash function be used instead of a true random mapping. Hash functions for this parallel context have been proposed and analyzed by Mehlhorn and Vishkin [17] (see also [5, 10]). These authors have suggested an elegant class of functions with some provably desirable properties: the class of polynomials of degree $O(\log p)$ in arithmetic modulo m, where p is the number of processors and m the total number of words in the memory space.

In this section it is observed that for hashing to succeed in parallel algorithms running at optimal efficiency some parallel slack is necessary, and a moderate amount is sufficient if g can be regarded as a constant.

To see necessity we note that if only p accesses are made in a superstep to p components at random, there is a high probability that one component will get about $\log p/\log\log p$ accesses, and some will get none. Hence, the machine will have to devote $\Omega(\log p/\log\log p)$ time units to this rather than just a constant, which would be necessary for optimal throughput. Logarithms to the base two are used here, as throughout this article.

The positive side is that if slightly more, namely $p \log p$, random accesses are made in a superstep, there is a high probability that each component will get not more than about $3 \log p$ which is only three times the expected number. Hence, these accesses could be implemented by the router in the optimal bound of $O(\log p)$. More generally, if $pf(p)$ accesses are made randomly for any function $f(p)$ growing faster than $\log p$, the worst-case access will exceed the average rate by even smaller factors.

This phenomenon can be exploited as follows. Suppose that each of the p components of the BSP computer consists of a memory and a processor. We make it simulate a parallel program with $v \geq p \log p$ virtual processors by allocating $v/p \geq \log p$ of them to each physical processor. In a superstep, the BSP machine simulates one step of each virtual processor. Then the v memory requests will be spread evenly, about v/p per processor, and hence the machine will be able to execute this superstep in optimal $O(v/p)$ time with high probability. This analysis assumes, of course, that the v requests are to distinct memory locations. The more general case of concurrent accesses will be considered in the next section. To keep the constants low the machine has to

be efficient both in hashing and in context switching.

The conclusion is that if hashing is to be exploited efficiently, the periodicity L may as well be at least logarithmic, and if it is logarithmic, optimality can be achieved. Furthermore, for the latter, known hash functions suffice (see [29]). In making this claim we are charging constant time for the overheads of evaluating the hash function even at runtime. In justifying this, we can take the view that evaluating the hash function can be done very locally and hence quickly. (The $O(\log \log p)$ parallel steps needed to evaluate the $\log p$ degree polynomials may then be regarded as being a constant.) Alternatively, we can hypothesize that hash functions that are easier to compute exist. For example, some positive analytic results are reported for constant degree polynomials in [1] and [22]. Indeed, currently there is no evidence to suggest that linear polynomials do not suffice. Besides ease of computation these have the additional advantage of mapping the memory space one-to-one. Finally, we note that the frequency of evaluating the addresses most often used can be reduced in practice by storing these addresses in tables.

Concurrent Memory Accesses on the BSPC

In the previous section we considered memory allocation in cases in which simultaneous accesses to the same memory location are not allowed. In practice it is often convenient in parallel programs to allow several processors to read from a location or to write to a location (if there is some convention for resolving inconsistencies), and to allow broadcasting of information from one to all other processors. A formal shared memory model allowing arbitrary patterns of simultaneous accesses is the concurrent read concurrent write (CRCW) PRAM (see [11]).

One approach to implementing concurrent memory accesses is by combining networks, networks that can combine and replicate messages in addition to delivering them point-to-point [8, 20]. In the BSP model, it is necessary to perform and charge for all the replicating and combining as processing operations at the components. It turns out, however, that even the most general model, the CRCW PRAM, can be simulated optimally on the BSP model given sufficient slack if g is regarded as a constant. In particular, it is shown in [28] that if $v = p^{1+\epsilon}$ for any $\epsilon > 0$, a v processor CRCW PRAM can be simulated on a p-processor BSP machine with $L \geq \log p$ in time $O(v/p)$ (where the constant multiplier grows as ϵ diminishes). The simulation uses a method for sorting integers in parallel due to Rajasekaran and Reif [19] and employed in a similar context to ours by Kruskal et al. [12]. Sorting is one of the basic techniques known for simulating concurrent accesses [4]. Since general sorting has non-linear complexity we need to limit the domain, in this case to integers, to have some chance of an optimal simulation.

The general simulation discussed above introduces constants that are better avoided where possible. Fortunately, in many frequently occurring situations much simpler solutions exist. For example, suppose that we are simulating v virtual processors on a p-processor BSP computer and know that at any instant at most h accesses are made to any one location. Then if $v = \Omega(hp \log p)$, concurrent accesses can be simulated optimally by simply replicating any data items that are to be sent to r locations r times at the source processor (and charging for their transmission as for r messages). Similarly, if any combining occurs, it does so at the target processor.

To show this works, we suppose that among the destination addresses of the v accesses made simultaneously there are t distinct ones, and the numbers going to them are $l_1, \ldots, l_t$ respectively, all at most h. Suppose these are scattered randomly and independently among p memory units. Then the probability that a fixed unit receives more than x accesses is the probability that the sum of t independent random variables $\eta_j (1 \leq j \leq t)$, each taking value l_j with probability p^{-1} and value 0 otherwise, has value more than x. But a corollary of a result of Hoeffding [9] (see [15]) is that if ξ_j are independent random variables $0 \leq \xi_j \leq 1$ with expectation $c_j (j = 1, \ldots, t)$ and μ is the mean of $\{c_j\}$ then for $\alpha < \min(\mu, 1 - \mu)$.

$$\text{Prob}\left(\sum_{i=1}^{t} \xi_i \geq (\mu + \alpha)t\right) \leq e^{-\alpha^2 t/3\mu}.$$

If we set $\xi_i = \eta_i/h$ so that $\mu = \Sigma l_j/(pht) = v/(pht)$, and let $\alpha = \mu$, then the probability of $2\mu t$ being exceeded is at most $e^{-\alpha t/3} = e^{-v/3ph} \leq p^{-\gamma}$ if $v \geq 3\gamma hp \log_e p$. Hence, the probability that among the p processors at least one receives more than twice the expected number of accesses is at most p times this quantity, or $p^{1-\gamma}$. Hence $\gamma > 1$ suffices to ensure optimality to within constant factors.

We also observe that several other global operations, such as broadcasting or the parallel prefix, that one might wish to have, are best implemented directly in hardware rather than through general simulations. The simulation result does imply, however, that for programs with sufficient slack these extra features provide only constant factor improvements asymptotically.

As the machine scales up the hardware, investment for communication needs to grow faster than that for computation.

BSP Algorithms without Hashing

Although the potential for automating memory and communication management via hashing is a major advantage of the model, the programmer may wish to retain control of these functions in order to improve performance or reduce the amount of

slack required in programming. It appears that for many computational problems, simple and natural assignments of memory and communication suffice for optimal implementations on the BSP model. Fox and Walker (see [30]) have suggested a portable programming environment based on a very similar observation. A systematic study of such *bulk-synchronous algorithms* remains to be done. We can, however, give some examples. We note that several other models of computation have been suggested—mostly on shared memory models—that allow for the extra costs of communication explicitly in some way. Several algorithms developed for these work equally well on the BSPC. Among such related models are the phase PRAM of Gibbons [7], which incorporates barrier synchronization in a similar way to ours, but uses a shared memory. Others include the delay model of Papadimitriou and Yannakakis [18], and the LPRAM of Aggarwal et al. [1].

The algorithms described below are all tightly synchronized in the sense that the runtime of their constituent subtasks can be predicted before runtime. There is also a context for parallelism where many tasks are to be executed with varying time requirements that cannot be determined in advance. In the most extreme case, one has a number of subtasks whose runtime cannot be predicted at all. In this general dynamic load-balancing situation there also exist phenomena that are compatible with barrier synchronization. In particular Karp has given a load-balancing algorithm that is optimal for any L for the model of Gibbons (see [7]).

The advantages of implementing algorithms directly on the BSP model (rather than compiling them automatically), increase as the bandwidth parameter g increases. Hence, it is appropriate to consider g explicitly in analyzing the performance of these algorithms. An algorithm in this model will be broken down into supersteps where the words read in each superstep are all last modified in a previous superstep. In a superstep of periodicity L, L local operations and a $\lfloor L/g \rfloor$ -relation message pattern can be realized. The parameters of the machine are therefore L, g and p the number of processors. Each algorithm also has as a parameter n, the size of the problem instance. The complexity of an algorithm can be expressed in several ways in terms of these parameters. We will describe parallel algorithms in which the time-processor product exceeds the number of computational operations by only a fixed multiplicative constant, independent of L, g, p and n, provided that L and g are below certain critical values. In such "optimal" algorithms there may still be several directions of possible improvements, namely in the multiplicative constant as well as in the critical values of g and L.

As a simple example of a tightly synchronized algorithm well suited for direct implementation, consider multiplying two $n \times n$ matrices A and B using the standard algorithm on $p \leq n^2$ processors. Suppose we assign to each processor the subproblem of computing an $n/\sqrt{p} \times n/\sqrt{p}$ submatrix of the product. Then each processor has to receive data describing $n/\sqrt{p}$ rows of A and $n/\sqrt{p}$ columns of B. Hence, each processor has to perform $2n^3/p$ additions and multiplications and receive $2n^2/\sqrt{p} \leq 2n^3/p$ messages. Clearly, if in addition each processor makes $2n^2/\sqrt{p}$ message transmissions, the runtime is affected by only a constant factor. Fortunately, no more than this number of transmissions is required even if the elements are simply replicated at source. This is because if the matrices A and B are initially distributed uniformly among the p processors, $2n^2/p$ elements in each, and each processor replicates each of its elements $\sqrt{p}$ times and sends them to the $\sqrt{p}$ processors that need these entries, the number of transmissions per processor will indeed be this $2n^2/\sqrt{p}$. This is an instance of the point made in the previous section, that concurrent accesses, when the access multiplicity h is suitably small, may be implemented efficiently by simply replicating data at the source. It is easily seen that optimal runtime $O(n^3/p)$ is achieved provided $g = O(n/\sqrt{p})$ and $L = O(n^3/p)$. (An alternative algorithm given in [1] that requires fewer messages altogether can be implemented to give optimal runtime with g as large as $O(n/p^{1/3})$ but L slightly smaller at $O(n^3/(p \log n))$.)

A case in which it would be inefficient to realize multiple accesses by replication at the source is broadcasting. Here, one processor needs to send copies of a message to each of n memory locations spread uniformly among p components. Sending one copy to each of the p components can be accomplished in $\log_d p$ supersteps by executing a logical d-ary tree. In each superstep, each processor involved transmits d copies to distinct components. Time $dg\log_d p$ is required for this. If $n/p-1$ further copies are made at each component, optimality (i.e., runtime $O(n/p)$) can be achieved if $d = O((n/(gp \log p)) \log (n/(gp \log p)))$ and $L = O(gd)$. The constraint on d clearly implies that $n = \Omega(gp \log p)$. Examples of these constraints are $g = 1$, in which case $n = p \log p$ and $L = O(1)$ are sufficient, and $g = \log p$, in which case $n = p(\log p)^2$ and $L = O(\log p)$ suffice.

An operation more powerful than broadcasting is parallel prefix [11, 13]. Given $x_1, \ldots x_n$, one needs to compute $x_1 \circ x_2 \circ \ldots \circ x_i$ for $1 \leq i \leq n$ for some associative operation $\circ$. The now-standard recursive algorithm for this, but with d-ary rather than binary recursion, yields exactly the same constraints as those obtained above for broadcasting.

There are several important algorithms such as the fast Fourier transform that can be implemented directly on the butterfly graph. As observed in [18], an instance of such a graph with n inputs can be divided into $(\log n)/\log d$ successive layers, where each layer consists of $(n \log d)/d$ independent butterfly graphs of $d/\log d$ inputs each. This is true for any $d \geq 2$ if the expressions are rounded to

integers appropriately. We can, therefore, evaluate such a graph on $p = (n \log d)/d$ processors in $(\log n)/\log d$ supersteps, in each of which each processor computes d local operations and sends and receives $d/\log d$ messages. Hence, optimality can be achieved if $g = O(\log d) = O(\log(n/p))$, and $L \leq d = O((n/p) \log (n/p))$.

A further problem for which bulk-synchronous algorithms are of interest is sorting. Among known algorithms that are well suited is Leighton's columnsort [14]. For sorting n items on $p = O(n^{1/3})$ processors it executes eight consecutive stages. In the odd-numbered stages each processor sorts a set of n/p elements sequentially. In the even-numbered stages, the data is permuted among the processors in a certain regular pattern. Hence, computation and communication are separated at the coarsest scale. For optimal runtime on the BSP model, the communication time $O(gn/p)$ must not exceed the computation time of $(n/p) \log (n/p)$ which is required by each stage of sequential comparison sorting. Hence, $g = O(\log(n/p))$ and $L = O((n/p) \log (n/p))$ suffice.

More generally, it is clear that any actual BSP machine would impose an upper bound on p, the number of processors, as well as a lower bound on the value of g that can be achieved. Also, for any g to be achieved, a lower bound on L may be implied. One can, therefore, imagine transportable BSP software to be written in a way that the code compiled depends not only on the problem size n but also on the parameters p, g and L.

Implementation on Packet Switching Networks

The communication medium or router of the BSP model is defined to be the simplest possible with the goal that it can be implemented efficiently in various competing technologies. In current parallel machines, the favored method of communication is via networks that do some kind of packet switching. Therefore, our main argument will refer to this. In

The advantages of implementing algorithms directly on the BSP model (rather than compiling them automatically), increase as the bandwidth parameter g increases.

implementing the BSP model on a packet switching network, the main tool available is that of pipelining communication. The conclusion will be that a network such as a hypercube will suffice for optimality to within constant factors, but only if its communication bandwidth is balanced with its computational capability. Thus, to simulate the BSP model with bandwidth factor g it is necessary that the computational bandwidth of a node does not exceed the communication bandwidth of the connection between a pair of nodes adjacent in the network by more than a factor of g.

Packet routing on regular networks has received considerable attention. Consider a hypercube network and suppose that in g units of time a packet can traverse one edge of it. Thus, a single packet will typically take $g \log p$ time to go to an arbitrary destination. A paradigmatic case of parallel packet routing is that of routing permutations. Here each of the p processors wishes to send a message to a distinct destination. What is required is a distributed routing algorithm that needs no global knowledge of the message pattern, and ensures that all the packets arrive fast, even when fully allowing for contention at the edges. It turns out that a simple two-phase randomized routing algorithm [26, 29] suffices to give runtime of about $2g \log p$ with overwhelming probability.

While this is optimal for permutation routing, it does not imply optimal BSP simulations immediately since it corresponds to the case of 1-relations and would require a factor of at least $\log p$ more in communication compared with computation time.

In order to obtain an optimal BSP simulation, we need to use the fact that two-phase randomized routing can support heavier message densities. It turns out that if there are $\log p$ packets initially at each node with at most $\log p$ destined to any one target, $O(g \log p)$ time still suffices for all the $p \log p$ packets to reach their destinations [25, 29]. In other words, $\log p$-relations can be realized essentially as fast as 1-relations. This gives an optimal simulation of a BSP machine with $L > g \log p$ since in each superstep we need to simulate L local operations at each processor and realize an L/g-relation in the router. All this can be simulated in time $O(L)$ on the hypercube. We note that the simulations give small constant factors and experiments show that small queues suffice.

Further details of results on routing can be found in [29]. All the indications are that this problem has a variety of practical and efficient solutions. For example, instead of store-and-forward message passing one could consider bit-streamed or wormhole routing which exhibit similar phenomena [2]. We also note that if the address space is already randomized by hashing, two-phase routing can be replaced by one-phase deterministic routing for implementing memory accesses [20].

Since the BSP model separates computation from communication, no particular network topology is favored beyond the requirement that a high throughput be delivered. An example related to the hypercube that suffices under similar conditions is the butterfly, which would consist of $(\log p) + 1$ levels of p nodes each. One of the levels would be allocated to processor/memory components and the rest to switches.

Implementation on Optical Crossbars

Since we envisage the BSP computer as being realizable in a variety of

technologies, we conclude here with the observation that it can be implemented optimally on a simple model of computation suggested by the possibilities of optical technology.

In this model, in each time step each of p components can transmit a message by directing a beam of light at a chosen component. If a component receives just one message it acknowledges it and transmission is considered successful. On the other hand, if more than one beam is directed at a node, none of the messages is successfully received at that node, and the absence of a valid acknowledgment informs a sender of the failure. Such a model has been considered in [3, 16].

In light of the earlier discussion on simulating shared memory by hashing using periodicity $L \geq \log p$, a crucial case for this optical model is that of a superstep in which each processor sends up to $\log p$ messages, each receives up to about the same number, and there is no other detectable pattern to the requested global communication. It is observed in [29] that a randomized algorithm of Anderson and Miller [3] can be adapted to perform this communication on this optical model in $O(\log p)$ time steps, which is optimal. Therefore, if such a time step corresponds to g time units, this model can simulate a $\Omega(p \log p)$ BSP computer optimally.

Conclusion

We have defined the BSP model and argued that it is a promising candidate as bridging model for general-purpose parallel computation. As supporting evidence, we have described how a variety of efficiency phenomena can be exploited by this one model. No single factor is, or can be, decisive in confirming the adequacy of a bridging model. It is the diversity of the considerations in support of the model and the apparent absence of contrary indications that are here most compelling.

The considerations we have analyzed are all concerned with providing guaranteed performance at near-optimal processor utilization. Since the primary object of parallel computing is to obtain high throughput, we consider such quantitative criteria to be critical. In the spectrum of imaginable computations we have addressed the end that is most communication intensive, since this case cannot be evaded in a general-purpose setting. We have been careful, however, to ensure that less-constrained computations, where independent processes can proceed with infrequent communication, are not penalized.

The model is intended to be below the language level and we hope that it is compatible with a variety of language styles. Several of our arguments, however, favor programs from which the compiler can efficiently abstract the necessary number of parallel streams of computation. Highly synchronized languages written in the PRAM style are clearly compatible. The model is consistent, however, with a number of other situations also. For example, in transaction processing where the length of transactions is statistically predictable a random allocation of processors would suffice.

The arguments given in this article in support of the BSP model are of three kinds. First it is argued that if the computational and communication bandwidth are suitably balanced (i.e., g is a small constant such as one) the model has a major advantage regarding programmability, as least for programs with sufficient slack. In that case the memory and communication management required to implement a virtual shared memory can be achieved with only a constant factor loss in processor utilization. The constants needed in the simulations are known to be small, except in the case that concurrent accesses are made with high levels of concurrency to each of many single locations simultaneously. We note that existing machines have higher values of g than is ideal for a BSP computer. The arguments of this article can be interpreted as saying that if the relative investment in communication hardware were suitably increased, machines with a new level of programmability would be obtained. We note that for certain programs in which automatic memory allocation is useful,the effective value of g can be made smaller than the physical value by exploiting locality and viewing the computation at a higher level of granularity. For example, in finite element methods the virtual memory can be regarded as partitioned into segments, each of which is to be stored in a single memory component. The number of computation steps per segment may then greatly exceed the number of nonlocal memory accesses.

The second kind of argument given in this article is that several important algorithms can be implemented directly on this model. Such an implementation avoids the overheads of automatic memory management and may exploit the relative advantage in throughput of computation over communication that may exist.

The third argument is that the BSP model can be implemented efficiently in a number of technologies. We illustrate this by giving an efficient simulation on a hypercube network as well as on a model suggested by optical communication. We note, however, that the BSP model is not particularly associated with any one technology or topology. The only requirement on the router is a certain level of communication throughput, however achieved. Clearly, the promise of optical technologies looks attractive in the BSP context. ◻

Acknowledgment

The author is grateful to an anonymous referee for several insightful suggestions concerning presentation.

References

1. Aggarwal, A., Chandra, A., and Snir, M. Communication complexity of PRAMs. *Theor. Comput. Sci.* To be published.
2. Aiello, B., Leighton, F.T., Maggs, B., and Neumann, M. Fast algorithms for bit-serial routing on a hypercube. *Manuscript*, 1990.
3. Anderson, R.J. and Miller, G.L. Optical communication for pointer based algorithms. Tech. Rep. CRI 88-14, Computer Science Dept., Univ. of Southern California, 1988.
4. Borodin, A. and Hopcroft, J.E. Routing merg-

ing and sorting on parallel models of computation. *J. Comput. Syst. Sci 30* (1985) 130–145.
5. Carter, J.L. and Wegman, M.N. Universal classes of hash functions. *J. Comput. Syst. Sci. 18* (1979) 143–154.
6. Eppstein, D. and Galil, Z. Parallel algorithmic techniques for combinatorial computation. *Annu. Rev. Comput. Sci. 3* (1988) 233–83.
7. Gibbons, P.B. A more practical PRAM model. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures.* (1989) pp. 158–168.
8. Gottlieb, A. et al. The NYU ultracomputer—Designing an MIMD shared memory parallel computer. *IEEE Trans. Comput. 32,* 2 (1983) 175–189.
9. Hoeffding, W. Probability inequalities for sums of bounded random variables. *Am. Stat. Assoc. J. 58* (1963) 13–30.
10. Karlin, A. and Upfal, E. Parallel hashing-An efficient implementation of shared memory. *J. ACM 35,* 4 (1988) 876–892.
11. Karp, R.M. and Ramachandran, V. A survey of parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science,* J. van Leeuwen, Ed. North Holland, Amsterdam, 1990.
12. Kruskal, C.P., Rudolph, L., and Snir, M. A complexity theory of efficient parallel algorithms. *Theor. Comput. Sci.* To be published.
13. Ladner, R.E. and Fischer, M.J. Parallel prefix computation. *J. ACM 27* (1980) 831–838.
14. Leighton, F.T. Tight bounds on the complexity of sorting. *IEEE Trans. Comput. C-34,* 4 (1985) 344–354.
15. Littlestone, N. From on-line to batch learning. COLT 89, Morgan Kaufmann, San Mateo, CA., (1989) 269–284.
16. Maniloff, E.S., Johnson, K.M., and Reif, J.H. Holographic routing network for parallel processing machines. Society of Photo Optical Instrumentation Engineers (SPIE), Paris, France 1989, V 1136, Holographic Optics II, Principles and Applications, 283–289.
17. Mehlhorn, K. and Vishkin, U. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Inf. 21* (1984) 339–374.
18. Papadimitriou, C.H. and Yannakakis, M. Towards an architecture-independent analysis of parallel algorithms. In *Proceedings of the Twentieth ACM Symposium on Theory of Computing* (1988) pp. 510–513.
19. Rajasekaran, S. and Reif, J.H. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput. 18,* 3 (1989) 594–607.
20. Ranade, A.G. How to emulate shared memory. In *Proceedings of the Twenty-eighth IEEE Symposium on Foundations of Computer Science* (1987) pp. 185–194.
21. Schwartz, J.T. Ultracomputers *ACM TOPLAS 2* (1980) 484–521.
22. Siegel, A. On universal classes of fast high performance hash functions. In *Proceedings of the Thirtieth IEEE Symposium on Foundations of Computer Science* (1989).
23. Snyder, L. Type architectures, shared memory, and the corollary of modest potential. *Annu. Rev. Comput. Sci. 1,* (1986) 289–317.
24. Turing, A.M. On computable numbers with an application to the Entscheidungs problem. In *Proceedings of the London Mathematical Society 42* 2 (1936) 230–265; correction, *ibidem 43* (1937) 544–546.
25. Upfal, E. Efficient schemes for parallel communication. *J. ACM 31,* 3 (1984) 507–517.
26. Valiant, L.G. A scheme for fast parallel communication. *SIAM J. Comput. 11* (1982) 350-361.
27. Valiant, L.G. Optimally universal parallel computers. Phil. Trans. R. Soc. Lond. A326 (1988) 373–376.
28. Valiant, L.G. Bulk-synchronous parallel computers. In *Parallel Processing and Artificial Intelligence,* M. Reeve and S.E. Zenith, Eds., Wiley, 1989 15–22.
29. Valiant, L.G. General purpose parallel architectures. In *Handbook of Theoretical Computer Science,* J. van Leeuwen, Ed., North Holland, Amsterdam 1990.
30. Walker, D.W. Portable programming within a message passing model: the FFT as an example. In Proc. 3rd Conference on Hypercube Concurrent Computers and Applications (1988), ACM Press.

CR Categories and Subject Descriptors: C.1.2. [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Parallel processors;* F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Parallelism.*
General Terms: Design
Additional Key Words and Phrases: Bulk-synchronous parallel model

About the Author:
Leslie G. Valiant is currently Gordon McKay Professor of Computer Science and Applied Mathematics at Harvard University. His research interests are in computational complexity, machine learning and parallel computation.

Author's Present Address: Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138; email: valiant@harvard.harvard.edu.

FINALLY APPLYING OBJECT-ORIENTED PROGRAMMING TO DATABASES

Object-Oriented Concepts, Databases, and Applications

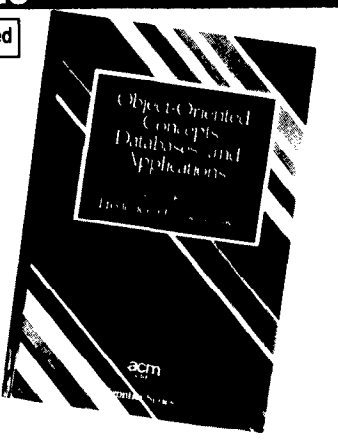
Satisfaction Guaranteed

Edited by WON KIM, MCC, and
FREDERICK H. LOCHOVSKY, University of Toronto

This wide-ranging introduction to the object-oriented paradigm begins by covering basic concepts and then moves on to a variety of applications in databases and other areas. Leading figures in the field wrote most of the contributions specifically for this volume.

Topics include object-oriented programming languages, application systems, operational object-oriented database systems, architectural issues, and directions in future research and development.

• ACM Press Books Frontier Series (A collaborative effort with Addison-Wesley) • Spring 1989 • 624 pp. • ISBN 0-201-14410-7 hardcover • Order Code 704892 • ACM Members $ 38.95 • Nonmembers $43.25



Reader Service #120

FOR FAST SERVICE CALL TOLL-FREE CREDIT CARD ORDERS ONLY 1-800-342-6626 (in AK & MD call 301-528-4261)

Yes, please send me the publication described above at the
☐ Member price ☐ Nonmember price. I am paying by:
☐ VISA ☐ MasterCard/Interbank ☐ American Express

Account # | | | | | | | | | | | | | | | | |

A $4.00 fee for each copy will be added for shipping and handling.

Signature _____ Exp. Date _____

I have enclosed a check for $_____ the total of my order. Because I've paid by check, shipping is free.

ACM Press
11 West 42nd St.
New York, NY 10036

acm PRESS Addison Wesley
ACM Press Books

Name _____
ACM Member # _____
Street Address _____
City/State/Zip _____
Phone (optional) _____

Book prices subject to change without notice. Allow 4-6 weeks for delivery.