A combining mechanism for parallel computers

A Combining Mechanism for Parallel Computers

Leslie G. Valiant

TR-24-92

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

A Combining Mechanism for Parallel Computers*

Leslie G. Valiant
Aiken Computation Laboratory
Harvard University
Cambridge, MA 02138

Abstract

In a multiprocessor computer communication among the components may be based either on a simple router, which delivers messages point-to-point like a mail service, or on a more elaborate combining network that, in return for a greater investment in hardware, can combine messages to the same address prior to delivery. This paper describes a mechanism for recirculating messages in a simple router so that the added functionality of a combining network, for arbitrary access patterns, can be achieved by it with provable efficiency. The method brings together the messages with the same destination address in more than one stage, and at a set of components that is determined by a hash function and decreases in number at each stage.

*This research was supported in part by a grant from the National Science Foundation, NSF-CCR-89-02500.

1 Introduction

A general purpose parallel computer needs to have a mechanism for realizing concurrent memory accesses efficiently. Several or all of possibly thousands of processors may wish to read the same memory address at the same time. Alternatively, several or all may wish to write a value into the same address, in which case some convention needs to be adopted about the desired outcome. In either case, the requests will have to converge from the various parts of the physical system to the one location.

If each request is sent directly to the component containing the relevant address, then this component will require time to handle them proportional to the number arriving there. In general, this becomes unacceptable if the number of processors p is large. This overhead, potentially linear in p, can be overcome by implementing the requests in more than one stage. In the first stage, for example, the requests to any one ultimate destination will converge in groups at various intermediate components, where each group is *combined* into a single request to be forwarded in the next stage. In the last stage all extant requests to an address finally converge to the chosen location. Thus the requests can be viewed as flowing from the leaves of a tree to the root. In some instances, as when the concurrent requests implement a read statement, a flow of information in the reverse direction, from the root to the leaves, needs to follow. In these instances, whenever requests are combined at a component, the sources of the requests in that stage are stored at that component, so that a complete record of the structure of the tree is maintained. In a general pattern of requests, accesses to several memory addresses may be present. In that case a combining tree has to be maintained for each address.

What is the most efficacious way of providing a multiprocessor computer with a combining facility that is acceptably efficient for the widest range of concurrent access patterns? In this paper we shall describe one proposed solution, and provide some analytic, experimental and, also, qualitative arguments in its favor.

In [14] we proposed the bulk-synchronous parallel (BSP) model of parallel computation in which the basic medium of inter-processor communication is a *router*, that delivers messages among the components point-to-point, but performs no combining or other computational operations itself. In this context a router means any device that can deliver a set of messages. It may be, for example, an optical device that transmits messages physically point-to-point. It was shown that shared memory with arbitrary concurrent accesses could be simulated on a p-processor BSP machine with only constant loss in efficiency asymptotically, if the simulated program had $p^{1+\varepsilon}$ fold parallelism for some positive constant ε. One important advantage of having as the communication medium this simplest option of being merely a message transmitter, is that it makes possible a competition for the highest throughput router among the widest possible range of technologies. In contrast, a medium that is required to perform more complex operations, such as combining (e.g. [4], [12], [13]) imposes more constraints on the technology. The crucial question is whether the extra capabilities of more complex hardware can be simulated in practice on the simple router, with acceptable loss of efficiency.

In this paper we lend support to the position that simple routers can indeed

implement concurrent accesses efficiently, by describing an algorithm for this that is apparently more efficient and practical than previous solutions [8], [15]. We give analytic results that show that $p^{1+1/m}$ requests to p units, with an arbitrary pattern of combining, can be realized in time $mp^{1/m}$ asymptotically as $p \to \infty$. A certain natural charging policy is used here for measuring time, and only the minimal assumptions are made on the uniformity of the requests among the components. Perhaps more significantly, the algorithm is of the form of a simple natural heuristic. Experiments suggest, for example, that for $p = 4096$ and with each processor making 32 requests, the cost of realizing arbitrary concurrency patterns as compared with patterns with no concurrency, is no more than a factor of about 3.5, even if nothing is known about the pattern. If the degree of concurrency is known then this factor can be made smaller.

We conclude that, when building a parallel computer, it may be efficacious to invest the bulk of the resources to be used for communication, in a simple router having maximal throughput. Although every general purpose parallel machine needs to have mechanisms for implementing arbitrary patterns of concurrent accesses, if, as it appears, difficult access patterns occur rarely enough, then our proposed mechanism for dealing with them is efficient enough that substantial investments in combining networks are not warranted.

2 Multi-Phase Combining

We consider a system consisting of p *components,* each of which has some memory and processing capabilities. A *(q,r)-pattern* among the p components is a set of communication requests in which each component sends at most q requests, each request has a destination that is a memory *address* in a particular component, and at most r requests are addressed to any one component. In this paper we shall charge $\max\{q, r\}$ units for executing directly a (q, r)-pattern on a router (as in the variant of the BSP model considered in [2].) This charging policy is intended to capture the basic idea that the requests made by any one component are processed sequentially as they are injected into the router, as are the requests arriving via the router at any one component. Thus q and r define the maximum cost of these two processes over all the components. Taking $\max\{q, r\}$ to be proportional to the overall time taken by the router is justified if the router has low latency, or if its latency is hidden by pipelining and its message load is high enough.

Distinct components contain disjoint sets of memory addresses. Several requests may share the same address (and therefore by implication also the same component.) We call the set of requests sharing the same address a *group.* The *degree* of a request is the number of elements in its group, and the degree of the pattern is the maximum degree of its elements. Thus if the pattern consists of n groups, of respective sizes $d_1, \cdots, d_n$ and destinations $t_1, \cdots, t_n$, then the *degree* of the pattern is $d = \max\{d_1, \cdots, d_n\}$.

The proposed *multi-phase combining* algorithm implements patterns of high degree by decomposing them into a sequence of patterns of low degree. At the end of

each phase, the requests having the same destination address that arrive at each component, are combined so that they can be transmitted as a single request in the phase to follow. For example, if every processor wishes to read the same word in memory then the requests form a $(1, p)$-pattern consisting of a single group, and has degree p. If implemented directly our charging policy would charge it p units of time. It can be decomposed, however, into a sequence of two $(1, \sqrt{p})-$patterns, each costing $\sqrt{p}$ units. The first allows each of $\sqrt{p}$ sets of $\sqrt{p}$ requests to converge to a separate component. Each of these components combines the $\sqrt{p}$ messages arriving, into one message, that is charged as one unit when it is sent on in the second phase. This second phase sends the $\sqrt{p}$ requests, that are so formed from the original p request, to their common destination. This decomposition into two phases, therefore, reduces the total charge from p to $2\sqrt{p}$ units. This example illustrates the software combining tree method proposed by Yew et al. [16] for dealing with a single hotspot.

For simplicity we shall assume that any two requests to the same destination address are combinable. This is true if, for example, we are executing at any one time either read or write statements, but not both. Our algorithm and analysis can be easily extended to cases in which more than one species of request cohabit.

We shall now describe our multi-phase algorithm for implementing arbitrary patterns on a simple router efficiently. For concreteness we shall describe the two instances that we analyzed, one of which we implemented. Many variants with comparable performance are possible and we shall discuss some of these in Section 5.

The algorithm has a *basis sequence* $(b_1, ..., b_m)$ of integers such that $\Pi_{i=1}^{m} b_i = p$. We also give analytic results that show for two variants of the algorithm. Suppose the space of possible addresses is denoted by M and that the components are numbered $\{0, ..., p-1\}$. Suppose also that $\{h_1, ..., h_m\}$ are hash functions where $h_i : M \rightarrow \{0, ..., b_1 \cdots b_i - 1\}$ and $\{k_1, ..., k_{m-1}\}$ are random functions where $k_i : \phi \rightarrow \{0, ..., b_{i+1} \cdots b_m - 1\}$. The important distinction is that for each pattern the hash functions h_i are chosen *once*, randomly from certain sets of hash functions, while the random functions k_i have values that are independently chosen randomly *at each invocation*, and do not depend on any argument. The i^{th} phase of the algorithm will at each component combine into one all the requests it has to any one destination t_j and send it to component $h_i(t_j)b_{i+1} \cdots b_m + k_i$. Note again that the requests to the same t_j will have the same value of $h_i(t_j)$, but will have k_i chosen randomly and independently for each of them at each component. It can be easily verified that, after i phases, among the requests destined for any t_j, sets of up to $b_1 \cdots b_i$ may have been combined into one. Also, if $i < m$, the resulting requests have been scattered randomly over $b_{i+1} \cdots b_m$ components, whose identities are determined by the hash function h_i. In particular after the last phase the request destined for t_j has been delivered to the hashed address $h_m(t_j)$. It turns out that the most promising mechanisms known for simulating a shared memory (PRAM) or BSP model on multiprocessor machines use a hashed address space (e.g. [14], [15]). Hence the algorithm as described implements hashing exactly as required in that context. As noted in Section 5, however, the algorithm can be adapted easily to deliver to physical rather than to hashed addresses.

An alternative algorithm with similar properties is obtained by replacing the ran-

dom function k_i in the above by the deterministic function $k_i^* : \{0, \cdots, p-1\} \rightarrow \{0, \cdots, b_{i+1} \cdots b_m - 1\}$ defined as $k_i^*(s) = s \bmod b_{i+1} \cdots b_m$, and in the i^{th} phase sending a request originating at component s and destined for address t_j to component $h_i(t_j)b_{i+1} \cdots b_m + k_j^*(s)$. This requires no randomization beyond the hash functions h_i, but evens out the load among the processors more slowly if the spread of the original requests is uneven.

As an illustration of these two versions of the algorithm, consider the following example consisting of 3 phases and having basis sequence (8,8,8), and, therefore, applying to the case $p = 512$. We represent the components as nine-bit binary numbers. The hash functions h_1, h_2 and h_3 applied to a destination *address* t take on values that are sequences of 3, 6 and 9 bits respectively. The packets destined for address t, that could initially start from all of the $8^3 = 512$ processors, will be directed to addresses having a common value of $h_1(t)$ in their first three bits. Hence these packets will have converged to at most $8^2 = 64$ processors after the first phase. The importance of having these 64 processors determined by a hash function, that treats distinct destination addresses as independently as possible, is that it prevents packets going to *distinct* addresses from creating hotspots. There remains the separate problem that the packets going to the one destination t must be spread sufficiently uniformly among these 64 processors that they do not create hotspots themselves. Our first solution is to spread them out by determining the last six bits in their addresses randomly and independently of each other, using a random function k_1. The second solution is to have as these last six bits the corresponding six bits of the source of the request. The latter is an attractive option for implementation in combination, for example, with $h_1(h)$ and $h_2(t)$ being prefixes of $h_3(t)$. The performance of the former solution is, however, independent of the source addresses. It is, therefore, the more appropriate for deriving experimental results that generalize. In either case the behavior of the second phase of the algorithm is similar to the first. The final phase brings all the packets destined for address t to processor $h_3(t)$. The effect of the overall algorithm is, therefore, to bring together the requests destined for one address in a tree of depth three, where the nodes have approximately equal degree and the processors that simulate them are chosen by a hash function in order to avoid hotspots.

3 Asymptotic Analysis

We shall establish the following property of the multi-phase algorithm, that holds asymptotically as the number of processors $p \rightarrow \infty$, for all patterns of all degrees.

Theorem *For any constant $\varepsilon > 0$, any integer $m \geq 1 + \lfloor \varepsilon^{-1} \rfloor$, and any constant $\delta > 0$, there is an m-phase algorithm that can realize any (q, r)-pattern with $q \leq p^{\varepsilon}$ in a number of steps that exceeds $(1 + \delta)mp^{\varepsilon}$ with probability less than $e^{-\Omega(p^{\varepsilon - 1/m})}$.*

The result assumes that the address space is hashed, as previously described, and for that reason does not depend on r. Also, the proof assumes that when choosing

the hash functions, we are choosing from a set of functions that allow the chosen one to behave randomly and independently for the various arguments at which it is evaluated.

The result as stated improves on the constant multiplier in the runtime of the best previously known method based on integer sorting [8], [15]. In particular the experimental results show that small values of δ can be attained with probability close to unity.

We shall use the following bound on the tail of the sum of independent random variables given in [9] and [10], and also derivable from [11].

Lemma *If $\xi_1, \cdots, \xi_n$ are independent random variables each taking values in the range $[0,1]$ such that the expectation of their sum is E, then for any $\delta > 0$,*

$$Prob\left(\sum_{i=1}^{n} \xi_i \geq (1+\delta)E\right) \leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{E} \quad .$$

Proof of Theorem

In the analysis we shall assume, for simplicity, that $b = p^{1/m}$ is an integer so that we can choose as basis sequence $(b_1, \cdots, b_m)$ where $b_1 = b_2 = \cdots = b_m = b$. Otherwise the b_i would be chosen so as to differ by at most one from each other. We consider how the algorithm behaves on an arbitrary (q,r)-pattern with $q \leq p^{\varepsilon}$, and with n destination addresses $t_1, \cdots, t_n$ and degrees $d_1, \cdots, d_n$, respectively. If we let $v = \sum_{i=1}^{n} d_i$ then clearly $v \leq qp \leq p^{1+\varepsilon}$.

At the start of phase i the j^{th} group of requests, namely those destined for t_j, will have been combined into at most

$$\min\{d_j, p/b^{i-1}\}$$

requests, since these number at most d_j at the start, and have converged to at most p/b^{i-1} processors by this time. For $i > 1$ they will be distributed randomly among the p/b^{i-1} components numbered $h_{i-1}(t_j)b^{m-i+1} + x$ for $0 \leq x < b^{m-i+1}$.

Now consider some fixed component numbered $yb^{m-i} + z$ where $0 \leq y < b^i$ and $0 \leq z < b^{m-i}$. Let η_j be the number of requests with destination t_j arriving at this component at the end of phase i. Then

$$Prob(\eta_j \geq u) \leq Prob(h_i(t_j) = y) \cdot B(\min\{d_j, p/b^{i-1}\}, b^i/p, u) \qquad -(1)$$

where $B(w, P, u)$ denotes the probability that in w independent Bernoulli trials, each with probability P of success, there are at least u successes. The first term gives the probability that the randomly chosen hash function h_i maps t_j to y, and equals b^{-i} clearly. The second bounds the probability that at least u of the requests are mapped by the invocations of the random function k_i, to the chosen value of z, and, by the Lemma above, can be upper bounded by

$$\left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{b} \quad \leq e^{-\Omega(p^{1/m})} \qquad -(2)$$

if $u = (1+\delta)b$ and $\delta > 0$, since the mean is at most $(p/b^{i-1})(b^i/p) = b$.

We shall now define new random variables $\xi_1, \cdots, \xi_n$ where

$$\xi_j = \begin{cases} \eta_j/((1+\delta)b) & \text{if } \eta_j \leq (1+\delta)b, \\ 1 & \text{otherwise.} \end{cases}$$

These satisfy the condition of the Lemma that $0 \leq \xi_j \leq 1$, and model the behavior of $\eta_1, \cdots, \eta_n$ exactly, except for the range $\eta_j \geq (1+\delta)b$, which is a very rare event by virtue of (2). Since, by (1), the expected value of η_j is at most $b^{-i} \cdot \min\{d_j, p/b^{i-1}\} \cdot b^i/p$, it follows from the definition of ξ_j that the sum of the expectations of $\xi_1, \cdots, \xi_n$ is

$$\begin{aligned} E &\leq \frac{1}{(1+\delta)b} \sum_{j=1}^{n} \frac{d_j}{p} \\ &\leq \frac{p^{1+\varepsilon}}{(1+\delta)bp} \leq \frac{p^{\varepsilon - 1/m}}{(1+\delta)}. \end{aligned}$$

Applying the Lemma to $\xi_1, \cdots, \xi_n$, assumed here to be independent, then gives that

$$\begin{aligned} Prob\left(\sum_{j=1}^{n} \xi_j \geq (1+\delta) \cdot \frac{p^{\varepsilon - 1/m}}{(1+\delta)}\right) &\leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{p^{\varepsilon - 1/m}/(1+\delta)} \\ &\leq e^{-\Omega(p^{\varepsilon - 1/m})} \qquad -(3) \end{aligned}$$

if $\delta > 0$. But, by the definition of ξ_j, the lefthand side is

$$Prob\left(\sum_{j=1}^{n} \eta_j \geq (1+\delta)p^{\varepsilon}\right) - \sum_{j=1}^{n} \mu_j \qquad -(4)$$

where μ_j is less than the probability that η_j exceeds $(1+\delta)b$, which by relations (1) and (2) is at most $e^{-\Omega(p^{1/m})}$. This is derived by partitioning the event referred to in the first term of (4) into two events according to whether or not $\eta_j \leq (1+\delta)b$ for every j. Hence we deduce from (3) and (4) that the probability that the number of requests $\sum_{j=1}^{n} \eta_j$ arriving at the chosen node at the end of phase i exceeds $(1+\delta)p^{\varepsilon}$ is still $e^{-\Omega(p^{\varepsilon - 1/m})}$, since the $n \leq p^{1+\varepsilon}$ choices of μ_j contribute only a lower order term in the exponent.

As there are p components and m phases, the probability that this charge is exceeded anywhere in the run is therefore pm times this same quantity, which is also $e^{-\Omega(p^{\varepsilon - 1/m})}$. Hence the result claimed in the Theorem follows.

For completeness we now prove the same result for the alternative algorithm in which in phase i any request originating at component s and destined for address t_j is sent to $h_i(t_j)b^{m-i} + k_i^*(s)$ where $k_i^*(s) = s \mod b^{m-i}$. Here at the start of

phase i the group of requests destined for t_j will have been combined into again at most $\min\{d_j, b^{m-i+1}\}$ requests, which are distributed among the b^{m-i+1} components numbered $h_{i-1}(t_j)b^{m-i+1} + x$ for $0 \leq x \leq b^{m-i+1}$. For any fixed component numbered $yb^{m-i} + z$ where $0 \leq y < b^i$ and $0 \leq z < b^{m-i}$ define η_j to be the number of requests with destination t_j arriving at this component at the end of phase i. Let X_j be the number of requests at the start of phase i destined for t_j from addresses with $k_i^*(s) = z$. Then clearly $X_j \leq b$, and $\sum X_j \leq p^\varepsilon b^i$ since only the b^i source components that coincide with z in the last bits contribute. Also $\eta_j = X_j$ with probability b^{-i} (i.e. if $h_i(t_j) = y$) and $\eta_j = 0$ otherwise. If we define $\xi_j = \eta_j/b$, so that $0 \leq \xi_j \leq 1$, then the sum of the expectations of ξ_j is

$$E \;\leq\; b^{-i-1} \sum_{j=1}^{n} X_j \;\leq\; b^{-i-1} p^\varepsilon b^i \;\leq\; p^{\varepsilon - 1/m}.$$

The result then follows from the Lemma as before.

4 Experimental Results

The multi-phase combining algorithm with the k_i chosen to be random functions was implemented as follows. In the basis sequence we used only powers of 2 (i.e. $b_i = 2^{a_i}$ for integers $a_i, 1 \leq i \leq m$). We used a pseudo-random number generator to generate new values of the functions k_i at each invocation. We also used a pseudo-random number generator to generate for each pattern the set of values $\{h_i(t_j) \mid 1 \leq i \leq m, 1 \leq j \leq n\}$. In particular we had the binary representation of $h_i(t_j)$ to be the prefix of the binary representation of $h_{i+1}(t_j)$, so that only a_{i+1} random new bits were chosen when determining the latter.

We ran experiments for the case $p = 2^{12}$ and $v = \sum_{j=1}^{n} d_j = 2^{12}$. We implemented (q,r)-patterns with $q = 32$, but with degrees varying from 2^{12} down to 1. Thus typically we had $n = 2^{17}/d$ groups each of degree d, for $d = 2^{12}, 2^{11}, \cdots, 2^0$. The reported results are all averages over 500 runs.

At one extreme we had 2^{17} groups of degree one and therefore no combining was required. The patterns were $(32,r)$-patterns where r depended on the maximum number of requests that the hashed address space placed into one component. The average value of r was determined experimentally to be 54.4. This is just the expected number of objects in the bucket having the most objects, if 2^{17} objects are placed randomly into 2^{12} buckets. Since this is the baseline performance of a pure router with a hashed address space, we computed the runtime of all our experiments as multiples of this basic unit, and call this multiple the *performance factor*.

At the other extreme we had 32 groups of degree 2^{12}, which corresponds to each of the components sending requests to the same set of 32 addresses. This requires the highest amount of combining.

We note that the $m = 1$ version of our algorithm (i.e. basis sequence (4096)), is the solution proposed in [14] for patterns of low degree. From Table 1 we see that if the degree is no more than the slack (*i.e.* $v/p = 32$) then the performance is indeed quite good, the performance factor being no worse than 3.7. This factor improves

rapidly as d decreases. On the other hand, the degree is clearly a lower bound on the runtime of the one-phase algorithm, and for $d = 2^{12}$ gives a performance factor greater than $4096/54.4 > 75$, which is unacceptable.

If the case $d = 1$ is implemented in several phases, then each phase performs hashing with no combining and contributes a factor of about 1 to the overall runtime. (The contribution is actually slightly more since we are charging $\max(q, r)$ rather than r for a (q,r)-pattern and irregularities in the distribution at the start of a phase contribute also.) This is also the case in early phases of the algorithm if d is small enough that little combining is done in that phase. In phases where much combining is done the performance factor can exceed one considerably. If the necessary combining is achieved in early phases, however, later phases may execute very fast since only few requests remain in the system. These phenomena can be discerned easily in Table 2, where for various values of d we give basis sequences that achieved factors below 3.

We note that the motivation for the charging policy in the BSP model is that some routers may achieve a satisfactory rate of throughput only when they have enough work to do (i.e. q is high enough when implementing a (q,r)-pattern) in one superstep. Hence the BSP model has a lower bound on the time for a superstep, determined by some parameters. Where this lower bound is relevant, we may give preference to basis-sequences that distribute the time cost evenly among the phases. On the other hand there are circumstances, for example, when the phases are implemented asynchronously as discussed below, when this issue does not arise.

Degree	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
Performance	82	53	35	20	13	8.1	5.3	3.7	2.6	1.9	1.5	1.2	1.0

Table 1. Performance factors for basis sequence (4096), i.e. one-phase, for various degrees for $p = 2^{12}$ and $v = 2^{17}$.

Degree	Basis Sequence	Phase 1	Phase 2	Phase 3	Phase 4	Performance
4096	(8,8,8,8)	1.51	.64	.37	.17	2.7
2048	(16,8,8,4)	1.60	.67	.38	.10	2.8
1024	(32,8,4,4)	1.71	.71	.26	.14	2.8
512	(32,8,4,4)	1.46	.92	.32	.16	2.8
256	(128,8,4)	1.84	.73	.20		2.8
128	(256,4,4)	1.90	.52	.25		2.7
64	(512,8)	1.95	.66			2.6
32	(1024,4)	1.98	.46			2.4
16	(1024,4)	1.58	.64			2.2

Table 2. Performance factors for various basis sequences for various degrees for $p = 2^{12}$ and $v = 2^{17}$. The factors are given separately for each phase, as well as in total.

Degree	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
(32,8,4,4)	3.4	3.0	2.8	2.8	3.0	3.2	3.4	3.6	3.8	4.0	4.1	4.1	4.1
(32,16,8)	3.4	3.1	3.0	3.1	3.3	3.3	3.3	3.4	3.4	3.4	3.3	3.2	3.0

Table 3. Performance factors for basis sequences (32,8,4,4) and (32,16,8) for various degrees for $p = 2^{17}$.

Tables 1 and 2 show that if we have information about the degrees of the patterns then we can find a good basis sequence that brings the performance factor of our algorithm below 3 in the whole range. If $d \leq 8$, then use of a single phase brings this factor below 2. The only assumption here is that the requests all have the same degree, which is the case we tested. If we have no knowledge about the degree of concurrency, then, as Table 3 shows, the basis sequences $(32, 8, 4, 4)$ and $(32, 16, 8)$ are good compromises and achieve performance factors of at most about 4 and 3.5 respectively throughout the whole range.

5 Variants of the Algorithm

The algorithm as described is "bulk-synchronized" in the sense that each phase has to finish before the next one starts. The correctness of the algorithm, however, does not require this. As each request in a phase arrives at a component, a check can be made to determine whether any other to the same address has been previously received, and if none has then the request can be sent on immediately to the next phase, without waiting for the previous phase to complete. Where it is permissible, such an *asynchronous* implementation can only improve performance. The actual performance in that case depends, however, on the order in which the router delivers the requests. Asynchrony may be introduced also if the requests are transmitted bit-serially.

The algorithm can be adapted to models of parallel computation other than the simple router. One candidate is what is called the S*PRAM in [15] that has been suggested as a model of various proposals for optical interconnects [1],[6]. Here at each cycle any component can transmit a message to any other, but only those receive messages that have just one targeted at them in that cycle. The senders find out immediately whether their transmission succeeded. Known general simulations of the BSP on the S*PRAM with slack $\log p$ or slightly more are known [3], [15] and these imply constant factor optimal implementations of our combining algorithm on the S*PRAM. There are clearly several possibilities for more efficient direct implementations also.

So far we have discussed versions of the algorithm that implement (q,r)-patterns in a hashed address space. The performance has been independent of the value of r because of the hashing. Suppose now that, we wish to send requests to physical addresses as, for example, when implementing a "direct BSP algorithm " [2]. We can clearly do this by sending the requests to hashed addresses first by the algorithm described, and then in one extra phase sending them to the correct destinations.

This last phase will be a pattern of degree 1, from randomly distributed sources. Also we can expect that the targets are distributed approximately uniformly among the components, since that is the purpose of using a direct algorithm. Hence this extra phase of the algorithm will run fast on the simple router. In particular, if the added last phase is a (q', r')-pattern, and the previous one is a (q'', r'')-pattern then clearly $q' \leq r''$. Also $r' \leq r^*$ where r^* is the maximum number of distinct destination addresses targeted in any component . Hence the cost $\max\{q', r'\}$ will be dominated by r'', which is controlled by randomization, and by r^* which is controlled by the programmer.

As an alternative to adding an extra phase to our multi-phase combining algorithm, we can also consider replacing its last phase by one that sends the requests directly to the actual rather than the hashed addresses. This will be efficient if the number of requests destined for each physical component, is small enough. When counting this number here, we have to allow for the multiplicity of each request group as defined by its degree in the last phase of the basic algorithm.

When implementing this multi-phase algorithm, provision has to be made by software or hardware or some combination, for storing at each phase the sources of the converging requests, so that this trace can be used for any necessary reverse flow of information. These provisions are also useful for implementing concurrent accesses when the decomposition of the pattern into phases is handcrafted by a programmer. This may be worthwhile for the sake of greater efficiency, for patterns that have a structure well-known to the programmer. Our algorithm, therefore, is also consistent with such direct implementations of concurrent accesses.

Finally, we note that our combining mechanism can be used for applications other than accesses. When requests are combined it is meaningful to perform almost any operation on their contents that is commutative and associative. The method can be used, for example, to find the sum, product, minimum, or Boolean disjunction over arbitrary sets of elements simultaneously.

References

[1] R.J. Anderson and G.L. Miller. Optical communication for pointer based algorithms. TR CRI-88-14, Computer Science Department, University of Southern California, 1988.

[2] A.V. Gerbessiotis and L.G. Valiant. Direct bulk-synchronous parallel algorithms. *Third Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science*, Vol 621, Springer-Verlag (1992) 1-18.

[3] M. Geréb-Graus and T. Tsantilas. Efficient optical communication in parallel computers. *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, June 29-July 1, (1992) 41-48.

[4] A. Gottlieb et al. The NYU Ultracomputer - Designing an MIMD shared-memory parallel computer. *IEEE Trans. on Computers*, C-32:2, (1983) 175-189.

[5] A. Hartmann and S. Redfield. Design sketches for optical crossbar switches intended for large scale parallel processing applications. Optical Engineering, *29*:3 (1989) 315-327.

[6] A. Karlin and E. Upfal. Parallel hashing – an efficient implementation of shared memory. *Proc. 18th ACM Symp. on Theory of Computing* (1986) 160-168.

[7] R.M. Karp and V. Ramachandran. A survey of algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science*, (J. van Leeuwen, ed.), North Holland, Amsterdam, (1990) 869-941.

[8] C.P. Kruskal, L. Rudolph and M. Snir. A complexity theory of efficient parallel algorithms. *Theor. Comp. Sci., 71* (1990) 95-132.

[9] N. Littlestone. Manuscript (1990).

[10] C. McDiarmid. On the method of bounded differences. Surveys in Combinatorics, 1989. J. Siemons (ed.) Lond. Math. Soc. Lecture Note Series *141* (1989) 148-188.

[11] P. Raghavan. Probabilistic construction of deterministic algorithms. *Proc. 27th IEEE Symp. on Foundations of Computer Science* (1986) 10-18.

[12] A. Ranade. How to emulate shared memory. *In Proc. 28th IEEE Symp. on Foundation of Computer Science* (1987) 185-194.

[13] H. Sullivan, T. Bashkow and D. Klappholtz. A large scale homogeneous fully distributed parallel machine. In *Proc. 4th Symp. on Computer Architecture* (1977) 105-124.

[14] L.G. Valiant. A bridging model for parallel computation. CACM *33* : 8 (1990) 103-111.

[15] L.G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), North Holland, Amsterdam (1990) 944-971.

[16] P.-C. Yew, N.-F. Tzeng and D.H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessers. *IEEE Trans. on Computers*, Vol. C-36:4 (1987) 388-395.