Practical Parallel Algorithms for Personalized Communication and Integer Sorting

David A. Bader* David R. Helman Joseph JáJá†

Institute for Advanced Computer Studies, and
Department of Electrical Engineering,
University of Maryland, College Park, MD 20742
{dbader, helman, joseph}@umiacs.umd.edu

November 28, 1995

Abstract

A fundamental challenge for parallel computing is to obtain high-level, architecture independent, algorithms which efficiently execute on general-purpose parallel machines. With the emergence of message passing standards such as MPI, it has become easier to design efficient and portable parallel algorithms by making use of these communication primitives. While existing primitives allow an assortment of collective communication routines, they do not handle an important communication event when most or all processors have non-uniformly sized personalized messages to exchange with each other. We focus in this paper on the **h-relation personalized communication** whose efficient implementation will allow high performance implementations of a large class of algorithms. While most previous h-relation algorithms use randomization, this paper presents a new deterministic approach for h-relation personalized communication. As an application, we present an efficient algorithm for stable integer sorting.

The algorithms presented in this paper have been coded in Split-C and run on a variety of platforms, including the Thinking Machines CM-5, IBM SP-1 and SP-2, Cray Research T3D, Meiko Scientific CS-2, and the Intel Paragon. Our experimental results are consistent with the theoretical analysis and illustrate the scalability and efficiency of our algorithms across different platforms. In fact, they seem to outperform all similar algorithms known to the authors on these platforms.

Keywords: Parallel Algorithms, Personalized Communication, Integer Sorting, Radix Sort, Communication Primitives, Routing h-Relations, Parallel Performance.

1 Problem Overview

A fundamental challenge for parallel computing is to obtain high-level, architecture independent, algorithms which efficiently execute on general-purpose parallel machines. This problem has become more tractable with the advent of message passing standards such as MPI [32], which seek to guarantee

*The support by NASA Graduate Student Researcher Fellowship No. NGT-50951 is gratefully acknowledged.
†Supported in part by NSF grant No. CCR-9103135 and NSF HPCC/GCAG grant No. BIR-9318183.

the availability of efficient implementations of certain basic collective communication routines. However, these proposed primitives are all regular in nature and exclude certain pervasive non-uniform communication tasks such as the **h-relation personalized communication**. In this problem, each processor has possibly different amounts of data to share with some subset of the other processors, such that each processor is the origin or destination of at most h messages. Clearly, such a task is endemic in parallel processing (e.g. [22, 44, 35]), and several authors have identified its efficient implementation as a prerequisite to efficient general purpose computing ([44]). In particular, in his "bridging model" for parallel computation, Valiant has identified the h-relation personalized communication as the basis for organizing communication between two consecutive major computation steps.

Previous parallel algorithms for personalized communication (typically for a hypercube, e.g. [28, 40, 36, 12, 13, 10, 1], a mesh, e.g. [24, 39, 14, 25], or other circuit switched network machines, e.g. [33, 19, 31, 37]) tend to be network or machine dependent, and thus not efficient when ported to current parallel machines. In this paper, we introduce a novel deterministic algorithm that is shown to be both efficient and scalable across a number of different platforms. In addition, the performance of our algorithm is invariant over the set of possible input distributions, unlike most of the published implementations.

As an application of this primitive, we consider the problem of sorting a set of n integers spread across a p-processor distributed memory machine, where $n \geq p^2$. Fast integer sorting is crucial for solving problems in many domains, and, as such, is used as a kernel in several parallel benchmarks such as NAS[1] [8] and SPLASH [46]. Because of the extensive and irregular communication requirements, previous parallel algorithms for sorting (a hypercube, e.g. [11, 1], or a mesh, e.g. [21, 30]) tend to be network or machine dependent, and therefore not efficient across current parallel machines. In this paper, we present an algorithm for integer sorting which couples the well known parallel radix sort algorithm together with our algorithm for personalized communication. We show that this sorting algorithm is both efficient and scalable across a number of different platforms.

Our algorithms are implemented in SPLIT-C [17], an extension of C for distributed memory machines. The algorithms make use of MPI-like communication primitives but do not make any assumptions as to how these primitives are actually implemented. The basic data transport is a **read** or **write** operation. The remote read and write typically have both blocking and non-blocking versions. Also, when reading or writing more than a single element, bulk data transports are provided with corresponding **bulk_read** and **bulk_write** primitives. Our collective communication primitives, described in detail in [7], are similar to those of MPI [32], the IBM POWERparallel [9], and the Cray MPP systems [16] and, for example, include the following: **transpose**, **bcast**, **gather**, and **scatter**. Brief descriptions of these are as follows. The **transpose** primitive is an all-to-all personalized com-

[1]Note that the NAS IS benchmark requires that the integers be ranked and not necessarily placed in sorted order.

munication in which each processor has to send a unique block of data to every processor, and all the blocks are of the same size. The **bcast** primitive is called to broadcast a block of data from a single source to all the remaining processors. The primitives **gather** and **scatter** are companion primitives whereby **scatter** divides a single array residing on a processor into equal-sized blocks which are then distributed to the remaining processors, and **gather** coalesces these blocks residing on the different processors into a single array on one processor. See [7, 6, 5] for algorithmic details, performance analyses, and empirical results for these communication primitives.

The organization of this paper is as follows. Section 2 presents our computation model for analyzing parallel algorithms. The Communication Library Primitive operations which are fundamental to the design of high-level algorithms are given in [7]. Section 3 introduces a practical algorithm for realizing h-relation personalized communication using these primitives. A parallel radix sort algorithm using the routing of h-relations is presented in Section 4. Finally, we describe our data sets and the experimental performance of our integer sorting algorithm in Section 5.

2 The Parallel Computation Model

We use a simple model to analyze the performance of our parallel algorithms. Each of our hardware platforms can be viewed as a collection of powerful processors connected by a communication network that can be modeled as a complete graph on which communication is subject to the restrictions imposed by the latency and the bandwidth properties of the network. We view a parallel algorithm as a sequence of local computations interleaved with communication steps, and we allow computation and communication to overlap. We account for communication costs as follows.

The transfer of a block consisting of m contiguous words, assuming no congestion, takes $\mathrm{O}(\tau + \sigma m)$ time, where τ is an upper bound on the latency of the network and σ is the time per word at which a processor can inject or receive data from the network. The cost of each of the communication primitives will be modeled by $\mathrm{O}(\tau + \sigma \max(m, p))$, where m is the maximum amount of data transmitted or received by a processor. Such cost (which is an overestimate) can be justified by using our earlier work [26, 27, 6, 7]. Using this cost model, we can evaluate the communication time $T_{comm}(n, p)$ of an algorithm as a function of the input size n, the number of processors p , and the parameters τ and σ. The coefficient of τ gives the total number of times collective communication primitives are used, and the coefficient of σ gives the maximum total amount of data exchanged between a processor and the remaining processors.

This communication model is close to a number of similar models (e.g. [18, 44, 2]) that have recently appeared in the literature and seems to be well-suited for designing parallel algorithms on current high performance platforms.

We define the computation time $T_{comp}(n, p)$ as the maximum time it takes a processor to perform all the local computation steps. In general, the overall performance $T_{comp}(n,p) + T_{comm}(n,p)$ involves a tradeoff between $T_{comm}(n,p)$ and $T_{comp}(n,p)$. Our aim is to develop parallel algorithms that achieve $T_{comp}(n,p) = O\left(\frac{T_{seq}}{p}\right)$ such that $T_{comm}(n,p)$ is minimum, where T_{seq} is the complexity of the best sequential algorithm. Such optimization has worked very well for the problems we have looked at, but other optimization criteria are possible. The important point to notice is that, in addition to scalability, our optimization criterion requires that the parallel algorithm be an efficient sequential algorithm (i.e., the total number of operations of the parallel algorithm is of the same order as T_{seq}).

3 An h-Relation Personalized Communication

Consider a set of n elements evenly distributed amongst p processors in such a manner that no processor holds more than $\frac{n}{p}$ elements. Each element consists of a pair $\langle data, dest \rangle$, where $dest$ is the location to where the $data$ is to be routed. There are no assumptions made about the pattern of data redistribution, except that no processor is the destination of more than h elements. We assume for simplicity (and without loss of generality) that h is an integral multiple of p.

A straightforward solution to this problem might attempt to sort the elements by destination and then route those elements with a given destination directly to the correct processor. No matter how the messages are scheduled, there exist cases that give rise to large variations of message sizes, and hence will result in an inefficient use of the communication bandwidth. Moreover, such a scheme cannot take advantage of regular communication primitives proposed under the MPI standard.

In our solution, we use two rounds of the **transpose** collective communication primitive. In the first round, each element is routed to an intermediate destination, and during the second round, it is routed to its final destination.

The pseudocode for our algorithm is as follows:

- **Step (1):** Each processor P_i, for $(0 \le i \le p-1)$, assigns its $\frac{n}{p}$ elements to one of p bins according to the following rule: if element k is the first occurrence of an element with destination j, then it is placed into bin $(i+j) \bmod p$. Otherwise, if the last element with destination j was placed in bin b, then element k is placed into bin $(b+1) \bmod p$.

- **Step (2):** Each processor P_i routes the contents of bin j to processor P_j, for $(0 \le i, j \le p-1)$. Since we will establish later that no bin can have more than $\frac{n}{p^2} + \frac{p}{2}$ elements, this is the equivalent to performing a **transpose** communication primitive with block size $\frac{n}{p^2} + \frac{p}{2}$.

- **Step (3):** Each processor P_i rearranges the elements received in **Step (2)** into bins according to each element's final destination.

- **Step (4):** Each processor P_i routes the contents of bin j to processor P_j, for $(0 \le i, j \le p-1)$. Since we will establish later that no bin can have more than $\frac{h}{p} + \frac{p}{2}$ elements, this is equivalent to performing a **transpose** primitive with block size $\frac{h}{p} + \frac{p}{2}$.

Correctness

To prove the correctness of our algorithm, we need to establish the bounds on the bin sizes claimed in **Steps (2)** and **(4)**. To establish the bound on the size of each bin in **Step (2)**, we note that the assignment process in this step is equivalent to sorting all the elements held in processor P_i by destination and then assigning all those elements with a common destination j one by one to successive bins[2], beginning with bin $(i+j) \bmod p$. Thus, the k^{th} element with destination j goes to bin $(i+j+k) \bmod p$. Let n_j be the number of elements a processor initially has with destination j. Notice that with this placement scheme, each bin will have at least $a_j = \lfloor \frac{n_j}{p} \rfloor$ elements with destination j, corresponding to the number of complete passes made around the bins, with $b_j = n_j \bmod p$ consecutive bins having one additional element for j. Moreover, this run of additional elements will begin from that bin to which we originally started placing those elements with destination j. This means that if bin l holds an additional element with destination j, the preceding $(l-(i+j)) \bmod p$ bins will also hold an additional element with destination j. Further, note that if bin l holds exactly q such additional elements, each such element from this set will have a unique destination. Since for each destination, the run of additional elements originates from a unique bin, for each distinct additional element in bin l, a unique number of consecutive bins preceding it will also hold an additional element with destination j. Consequently, if bin l holds exactly q additional elements, there must be a minimum of $1 + 2 + 3 + ... + (q-3) + (q-2) + (q-1)$ additional elements in the bins preceding bin l for a minimum total of $\frac{q}{2}(q+1)$ additional elements distributed amongst the p bins.

Consider the largest bin which holds $A = \sum_{j=0}^{p-1} a_j$ of the evenly placed elements and δ of the additional elements, and let its size be $binsize = A + \delta$. Recall that if a bin holds δ additional elements, then there must be at least $\frac{\delta}{2}(\delta+1)$ additional elements somehow distributed amongst the p bins. Thus,

$$\begin{aligned} \frac{n}{p} &= \sum_{j=0}^{p-1} n_j = p \sum_j a_j + \sum_j b_j \\ &\geq pA + \frac{\delta}{2}(\delta + 1). \end{aligned} \tag{1}$$

Rearranging, we get

$$A \leq \frac{n}{p^2} - \frac{\delta}{2p}(\delta + 1). \tag{2}$$

Thus, we have that

$$binsize \leq \frac{n}{p^2} - \frac{\delta}{2p}(\delta + 1) + \delta. \tag{3}$$

[2]The successor of bin $p-1$ is bin 0.

Since the right hand side of this equation is maximized over $\delta \in \{0, \ldots, p-1\}$ when $\delta = p-1$, it follows that

$$binsize \leq \frac{n}{p^2} + \frac{p-1}{2}. \;\; \Box \tag{4}$$

One can show that this bound is tight as there are cases for which the upper bound is achieved.

To bound the bin size in **Step (4)**, recall that the number of elements in bin j at processor i is simply the number of elements which arrive at processor i in **Step (2)** which are bound for destination j. Since the elements which arrive at processor i in **Step (2)** are simply the contents of the i^{th} bins formed at the end of **Step (1)** in processors 0 through $p-1$, bounding **Step (4)** is simply the task of bounding the number of elements marked for destination j which are put in any of the p i^{th} bins in **Step (1)**. For our purposes, then, we can think of the concatenation of these p i^{th} bins as being one *superbin*, and we can view **Step (1)** as a process in which each processor deals its set of n_j elements bound for destination j into p possible *superbins*, each beginning with a unique *superbin* $(i+j) \bmod p$. This is precisely the situation considered in our analysis of the first phase, except now the total number of elements to be distributed is at most h. By the previous argument, the bin size for the second phase is bounded by

$$binsize \leq \frac{h}{p} + \frac{p-1}{2}. \;\; \Box \tag{5}$$

Overall Complexity of the Algorithm

Clearly, all computation in this algorithm can be performed in $T_{comp}(n,p) = \mathrm{O}(h)$. The **transpose** primitive, whose analysis is given in [7], takes $T_{comm}(n,p) \leq \tau + \left(\frac{n}{p^2} + \frac{p}{2}\right)(p-1)\sigma$ in the second step, and $T_{comm}(n,p) \leq \tau + \left(\frac{h}{p} + \frac{p}{2}\right)(p-1)\sigma$ in the last step. Thus, the overall complexity of our algorithm is given by

$$\begin{aligned} T(n,p) &= T_{comp}(n,p) + T_{comm}(n,p) \\ &= \mathrm{O}\Big(h + \tau + \left(h + \tfrac{n}{p} + p^2\right)\sigma\Big) \\ &= \mathrm{O}\Big(h + \tau + (h + p^2)\,\sigma\Big), \end{aligned} \tag{6}$$

for $p^2 \leq n$. Clearly, the local computation bound is asymptotically optimal. As for the communication bound, $\tau + \left(h + \frac{n}{p}\right)\sigma$ is a lower bound as in the worst case a processor sends $\frac{n}{p}$ elements and receives h elements.

A similar two-stage algorithm for personalized communication which appeared after an earlier draft of this manuscript ([4]) is the transportation primitive (TP) by Ranka et al. ([34, 38]). It is important to note the differences between the TP and our algorithm. First, our algorithm is completely deterministic. Second, our intermediate communication steps use the **transpose** operation, which is extremely important for high performance, since many parallel machines ([9, 16]) and standards, e.g.

MPI [32], offer low-level library implementations for the **transpose** primitive. Third, we use less overhead. For example, the TP requires an initial $O(p^2)$ communication step to distribute a global control matrix, to calculate statistics from this control matrix, and to communicate with irregular buffer sizes. In contrast, our routing algorithm functions in an entirely distributed fashion, with no need for global control structures.

Despite its simplicity, our algorithm compares favorably with those developed recently by other researchers. For example, Gerbessiotis and Valiant [22] reported that they could route an h-relation using $2\frac{h}{p}(1+o(1)) + O(\log p)$ total exchange rounds with high probability for $h = \omega(p \log p)$ [3], where they define a total exchange as the **transpose** of a $p \times p$ array. Rao et al. [35] improved this bound to $2\frac{h}{p}(1+o(1)) + O(\log^* p)$ total exchange rounds with high probability and for $h = \omega(p \log^* p)$. By contrast, our deterministic algorithm requires at most $2\frac{h}{p}(1+o(1))$ of their total exchanges for $h = \omega(p^2 \log^* p)$.

3.1 Experimental Results

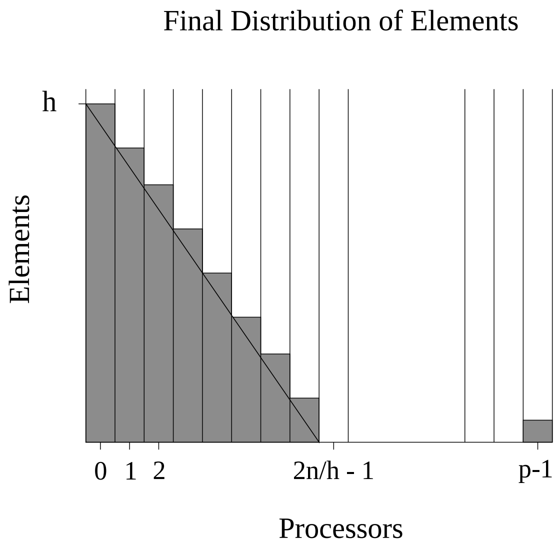


Figure 1: Final distribution of the keys corresponding to our input data sets

We examine the performance of our h-relation algorithm on various configurations of the TMC CM-5, IBM SP-2, Meiko CS-2, and Cray T3D, using four values of h: $h = \frac{n}{p}$, $2\frac{n}{p}$, $4\frac{n}{p}$, and $8\frac{n}{p}$. The data sets used in these experiments are defined as follows. Our input of size n is initially distributed cyclically across the p processors such that each processor P_i initially holds $\frac{n}{p}$ keys, for $(0 \le i \le p-1)$. For $h = \frac{n}{p}$ the input consists of $v_0 = \frac{n}{p}$ keys labelled for P_0, followed by $v_1 = \frac{n}{p}$ keys labelled for P_1, and so forth, (with $v_i = \frac{n}{p}$ keys labelled for P_i), with the last $v_{p-1} = \frac{n}{p}$ keys labelled for P_{p-1}. Note that

[3] Note that $f(n) = \omega(g(n))$ iff $g(n) = o(f(n))$.

this results in the same data movement as the **transpose** primitive[4]. For $h > \frac{n}{p}$, instead of an equal number of elements destined for each processor, the function v_i, for $(0 \le i \le p-1)$, is characterized by

$$\begin{cases} \left\lfloor h\left(1 - \frac{h}{2n-h} i\right)\right\rfloor, & \text{if } i < \frac{2n}{h},\ i \neq p-1, \\ 0, & \text{if } \frac{2n}{h} \le i < p-1, \\ n - \sum_{j=0}^{\frac{2n}{h}-1} \left\lfloor h\left(1 - \frac{h}{2n-h} j\right)\right\rfloor, & \text{if } i = p-1. \end{cases} \tag{7}$$

The result of this data movement, shown in Figure 1, is that processor 0 receives the largest imbalance of elements, i.e. h, while other processors receive varying block sizes ranging from 0 to at most h. For $h = 8\frac{n}{p}$, approximately $\frac{3p}{4}$ processors receive no elements, and hence this represents an extremely unbalanced case. Note that in these tests, each element consists of two integer[5] fields, *data* and *dest*, although only the destination field *dest* is used to route each element.

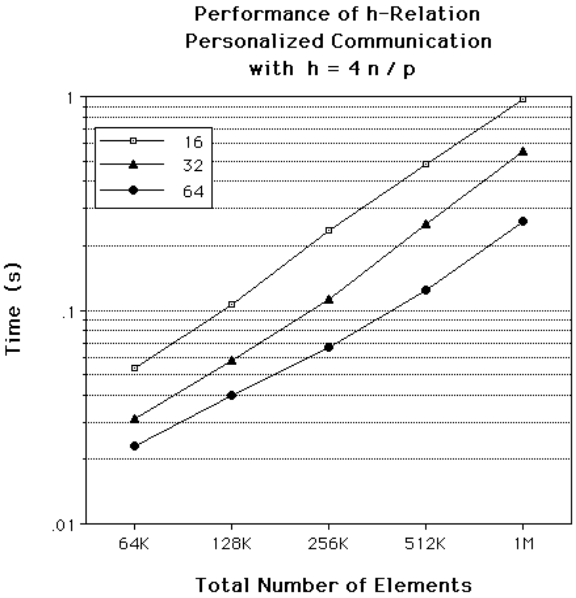
As shown in Figure 2, the time to route an h-relation personalized communication for a given input size on a varying number of processors (p) scales inversely with p whenever n is large compared with p. For small inputs compared with the machine size, however, the communication time is dominated by $O(p^2)$ as shown in the case of the 128-processor Cray T3D with $n = 256K$. The routing time for a fixed problem and machine size varies directly with the parameter h (see Figure 6 in Appendix A). These empirical results from a variety of parallel machines are consistent with the analysis given in Eq. (8). We have used vendor-supplied libraries for collective communication primitives on the IBM SP-2 implementation. The other machines used in this experiment do not have vendor-supported collective communication libraries, and hence we used our generic communication primitives as described in [7, 6, 5].

3.2 Comparison with Single-Phase Algorithms

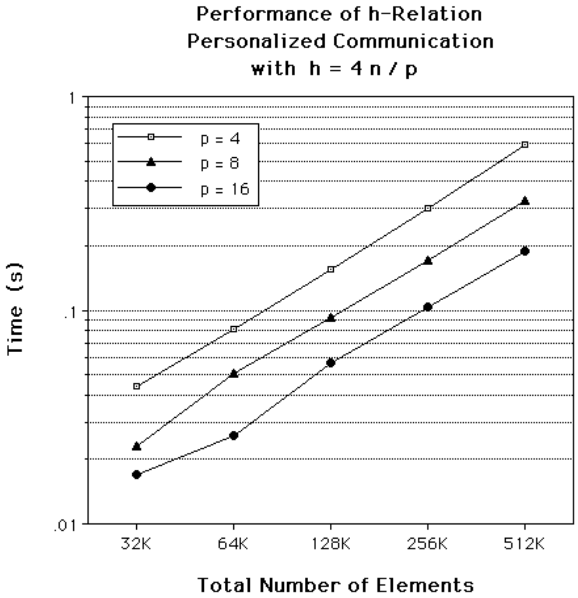
It has been widely believed that an efficient algorithm for personalized communication is a *single-phase* algorithm in which data travels directly from source to destination with no intermediate routing. These single-phase algorithms generally partition messages into contention-free routing steps separated by global synchronizations. As far as we can tell, this algorithm was first reported (in Japanese) by Take ([40]) for the hypercube network topology. Later, several variations of this algorithm were developed (still dependent upon network topology) such as the Optimal Circuit Switched, Hypercube, or Mesh Algorithm ([37, 10, 25, 36, 12, 13, 14, 1, 31, 23, 33, 19, 24]), the Pairwise-Exchange (PEX) algorithm ([43, 41, 42]), and the general Linear Permutation algorithm ([45]). For our comparison, we consider the standard algorithm consisting of p steps, such that during step i, $(0 \le i \le p-1)$, processor j sends data labelled for processor $k = i \oplus j$ directly to P_k. Figure 3 presents the

[4] Note that the personalized communication is more general than a **transpose** primitive and does not make the assumption that data is already held in contiguous, regular sized buffers.

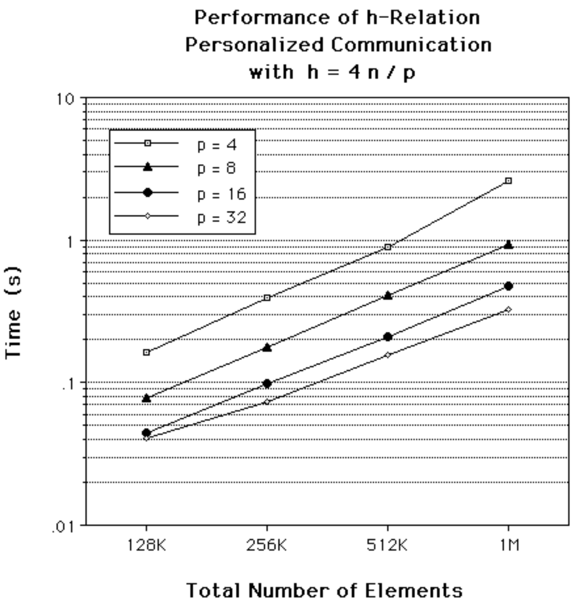
[5] In all our test machines, an integer is 4-bytes, except the Cray T3D, where an integer is 8-bytes.



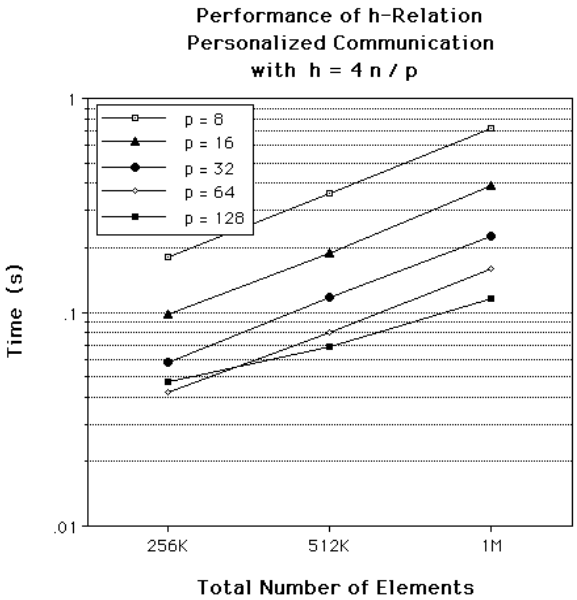
TMC CM-5



IBM SP-2

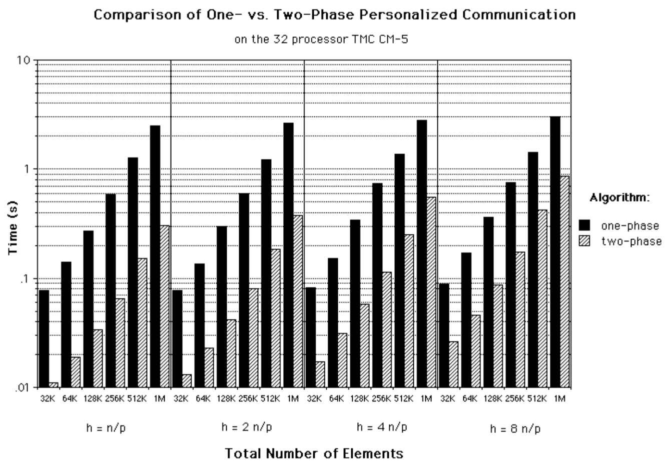


Meiko CS-2

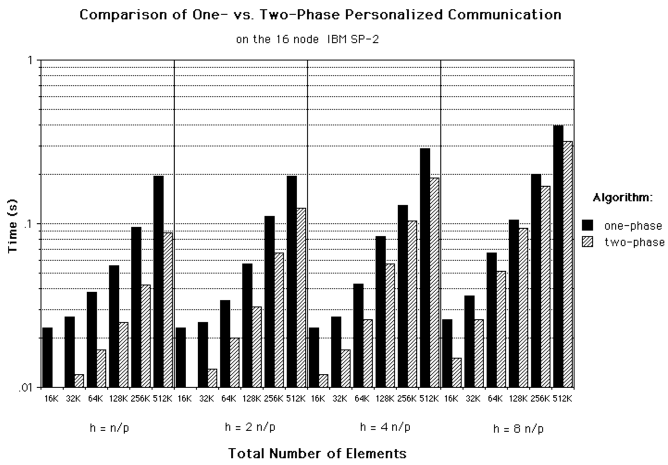


Cray Research T3D

Figure 2: Performance of personalized communication ($h = 4\frac{n}{p}$) with respect to machine and problem size



TMC CM-5



IBM SP-2

Figure 3: Comparison of one- and two-phase personalized communication algorithms

results of our comparison, providing empirical support for the notion that our two-phase personalized communication scheme is faster than the single-phase communication algorithm described above.

3.3 General Case

We now consider the general case in which each processor is the source of at most h_1 elements and the destination of at most h_2 elements. We can use the same deterministic algorithm with the block size of the **transpose** in **Step (2)** being $\frac{h_1}{p} + \frac{p}{2}$ and the block size of the **transpose** in **Step (4)** being $\frac{h_2}{p} + \frac{p}{2}$. The resulting overall complexity is $O\Big(h_1 + h_2 + \tau + (h_1 + h_2 + p^2)\,\sigma\Big)$. Alternatively for large variances ($h_1 \gg h_2$), we can use our dynamic data redistribution algorithm in [7] followed by our deterministic algorithm described earlier. The resulting overall complexity will also be the same.

4 Parallel Integer Sorting

Consider the problem of sorting n integer keys in the range $[0, M-1]$ that are distributed equally over a p-processor distributed memory machine. An efficient algorithm is **radix sort** that decomposes each key into groups of r-bit blocks, for a suitably chosen r, and sorts the keys by sorting on each of the r-bit blocks beginning with the block containing the least significant bit positions [29]. Let $R = 2^r \geq p$. Assume (w.l.o.g.) that the number of processors is a power of two, say $p = 2^k$, and hence $\frac{R}{p}$ is an integer $= 2^{r-k} \geq 1$. Our algorithm demonstrates efficient uses of the **transpose** communication primitive, as well as the h-relation communication scheme.

4.1 Counting Sort Algorithm

We start by describing the counting sort algorithm used to sort on individual blocks of the keys. The **Counting Sort** algorithm sorts n integers in the range $[0, R-1]$ by using R counters to accumulate the number of keys equal to i in bucket B_i, for $0 \le i \le R-1$, followed by determining the rank of the each element. Once the rank of each element is known, we can use our h-relation personalized communication to move each element into the correct position; in this case $h = \frac{n}{p}$. Counting Sort is a **stable** sorting routine, that is, if two keys are identical, their relative order in the final sort remains the same as their initial order.

In a practical integer sorting problem, we expect $R \approx \frac{n}{p}$. The pseudocode for our Counting Sort algorithm uses six major steps and is as follows.

- **Step (1):** For each processor i, count the frequency of its $\frac{n}{p}$ keys; that is, compute $I[i][k]$, the number of keys equal to k, for $(0 \le k \le R-1)$.
- **Step (2):** Apply the **transpose** primitive to the I array using the block size $\frac{R}{p}$. Hence, at the end of this step, each processor will hold $\frac{R}{p}$ consecutive rows of I.
- **Step (3):** Each processor locally computes the prefix-sums of its rows of the array I.
- **Step (4):** Apply the **(inverse) transpose** primitive to the R corresponding prefix-sums augmented by the total count for each bin. The block size of the **transpose** primitive is $2\frac{R}{p}$.
- **Step (5):** Each processor computes the ranks of local elements.
- **Step (6):** Perform a personalized communication of keys to rank location using our h-relation algorithm for $h = \frac{n}{p}$.

The analysis of our counting sort algorithm is as follows.**Steps (1)**, **(3)**, and **(5)** execute in $O\left(\frac{n}{p} + R\right)$ local computation time with no communication. **Steps (2)**, **(4)**, and **(6)** are communication supersets and have the following analysis. **Steps (2)** and **(4)** are the **transpose** primitive with block sizes $\frac{R}{p}$ and $2\frac{R}{p}$ and hence result in $O(\tau + R\sigma)$ communication. **Step (6)** uses the personalized communication primitive for n elements distributed equally across p processors. Because this routing is a permutation ($h = \frac{n}{p}$), it has the following complexity

$$T(n,p) = O\left(\frac{n}{p} + \tau + \left(\frac{n}{p} + p^2\right)\sigma\right) \tag{8}$$

provided that $p^2 \le n$. Thus, the overall complexity of our Counting Sort algorithm is given by

$$\begin{aligned} T(n,p) &= T_{comp}(n,p) + T_{comm}(n,p) \\ &= O\left(\frac{n}{p} + R + \tau + \left(R + \frac{n}{p} + p^2\right)\sigma\right). \end{aligned} \tag{9}$$

Notice that an obvious lower bound to sort the integers is $\Omega\left(\frac{n}{p} + \tau + \frac{n}{p}\sigma\right)$, and hence our algorithm is asymptotically optimal when $R = O\left(\frac{n}{p}\right)$ and $p^3 = O(n)$.

4.2 Radix Sort Algorithm

Radix Sort makes several passes of the previous Counting Sort in order to completely sort integer keys. Counting Sort can be used as the intermediate sorting routine because it provides a stable sort. Let the n integer keys fall in the range $[0, M-1]$, and $M = 2^b$. Then we need $\frac{b}{r}$ passes of Counting Sort; each pass works on r-bit blocks of the input keys, starting from the least significant block of r bits to the most significant block. Therefore, the overall complexity of Radix Sort is exactly $\frac{b}{r}$ times that of Counting Sort. We choose the radix R to be $\frac{n}{p}$ (note that we are assuming $p^2 \leq n$), and a typical value is $R = 1024$. Assuming that M is polynomial in n, $\frac{b}{r}$ becomes a constant, and therefore, the total complexity reduces to $T(n,p) = O\left(\frac{n}{p} + \tau + \left(\frac{n}{p} + p^2\right)\sigma\right)$. Thus, the computational analysis derived for radix sort is asymptotically optimal since sequential radix sort runs in $\Theta(n)$ whenever the range of integers is polynomial in n. The lower bound for communication is $\tau + \frac{n}{p}\sigma$ since each processor might need to inject all of its elements into the network, and the communication complexity is asymptotically optimal whenever $p^3 = O(n)$.

5 Performance Evaluation of Radix Sort

5.1 Data sets

Five input distributions are used to test our integer sorting algorithm.

- **[R]:** random integers with entropy of 31 bits per key[6];
- **[S]:** random integers with entropy of 6.2 bits per key[7];
- **[C]:** keys are consecutive in value (from 0 to $n-1$) and are placed cyclically across the processors;
- **[N]:** this input is taken from the NAS Parallel Benchmark for Integer Sorting [8]. Keys are integers in the range $[0, 2^{19})$, and each key is the average of four consecutive uniformly distributed pseudo-random numbers generated by the following recurrence:

$$x_{k+1} = ax_k \pmod{2^{46}}$$

  where $a = 5^{13}$ and the seed $x_0 = 314159265$. Thus, the distribution of the key values is a Gaussian approximation. On a p-processor machine, the first $\frac{n}{p}$ generated keys are assigned to P_0, the next $\frac{n}{p}$ to P_1, and so forth, until each processor has $\frac{n}{p}$ keys.

[6] Entropy of 31 implies that keys values are uniformly distributed in the range $[0, 2^{31})$.

[7] Entropy of 6.2 implies that each key is the result of the bitwise-AND boolean operation applied to five successive keys of entropy 31.

5.2 Experimental Results: Radix Sort

For each experiment, the input contains a total of $n = 2^d$ integers distributed evenly across p processors. The output consists of the sorted elements held in an array congruent with the input. Each processor's output block of elements is in non-descending order, and no element in processor i is greater than any element in processor j, for all $i < j$. Note that we use 32-bit keys and sort using all 32-bits, even when the input distribution is known to be more restrictive, such as the **N** input which contains only 19 significant bits.

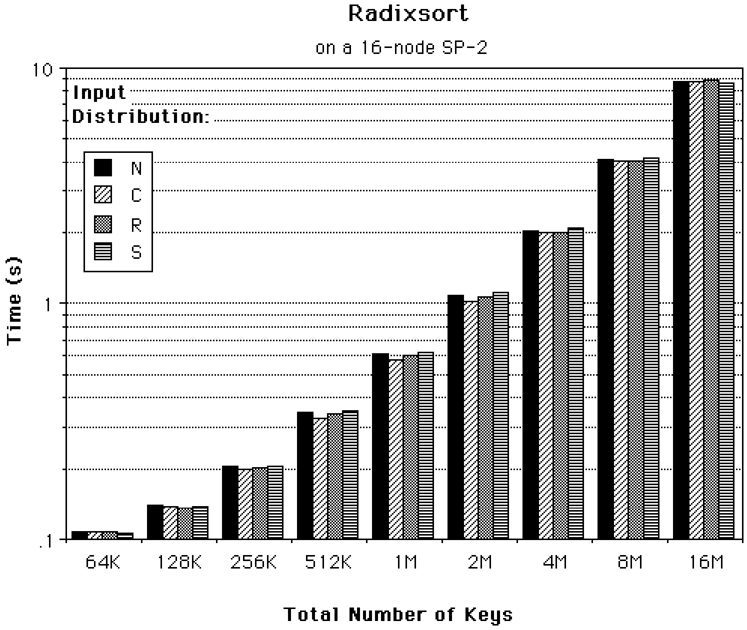
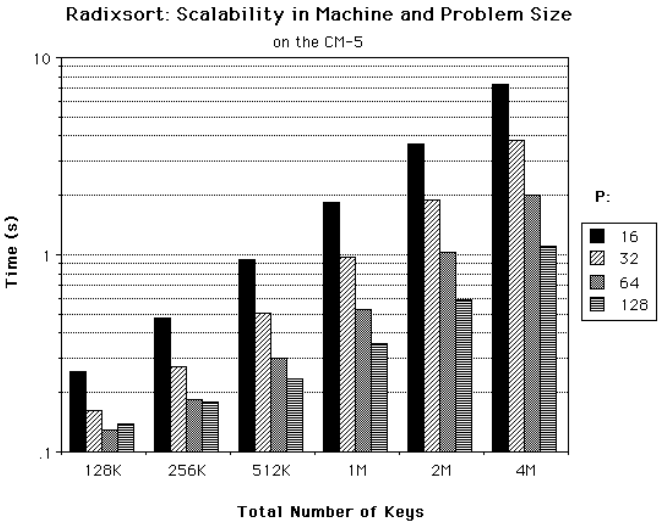


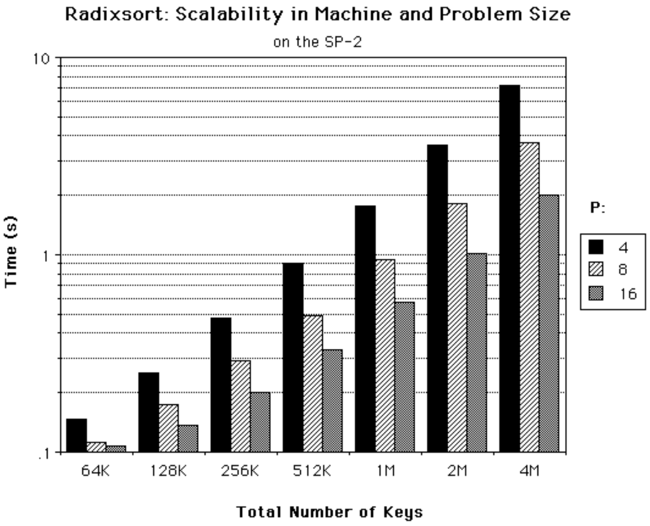
Figure 4: Performance is independent of key distribution

The performance of our radix sort is independent of input distribution, as shown in Figure 4.. This figure presents results from the IBM SP-2; results obtained from other machines, such as the CM-5, CS-2, and T3D, validate this claim as well.

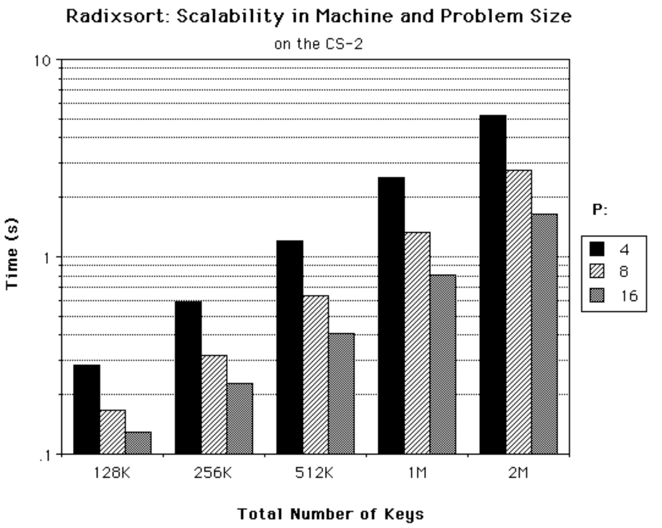
As shown in Figure 5, the execution time of radix sort using a fixed number of processors is linear in input size n. Note that this figure is a log-log plot. Since $\frac{b}{r}$ and R are constants for a given problem size, the running time is $O\left(\frac{n}{p}\right)$, validating our prediction from the bounds in the previous section. In addition, the execution time of radix sort for a given input size on a varying number of processors (p) scales inversely with p. Again, this was predicted by our earlier analysis.



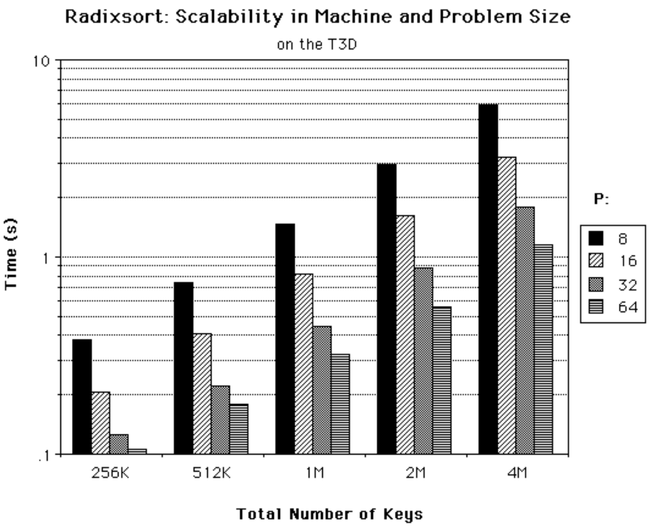
TMC CM-5



IBM SP-2



Meiko CS-2



Cray Research T3D

Figure 5: Scalability of Radix Sort With Respect to Machine and Problem Size

5.3 Comparison with Other Implementations

Table I presents a comparison of our radix sort with that of an implementation using only single phase routing by Alexandrov et al.[8] which we will refer to as the AIS code. Performance of the latter code, which had been optimized for the Meiko CS-2, is given in [2]. Note that the AIS implementation is based upon the original version by Dusseau ([20, 18]). Also, all codes in this comparison have been written in the SPLIT-C language [17]. Our algorithm is referred to as BHJ.

Input	SP-2 $p = 16$		CM-5 $p = 32$	
	[AIS+95]	[BHJ95]	[AIS+95]	[BHJ95]
[R], $\frac{n}{p} = 4K$	0.474	0.107	1.63	0.163
[R], $\frac{n}{p} = 64K$	0.938	0.592	3.41	1.91
[R], $\frac{n}{p} = 512K$	4.13	4.03	19.2	15.1
[C], $\frac{n}{p} = 4K$	0.479	0.107	1.64	0.163
[C], $\frac{n}{p} = 64K$	0.958	0.584	3.31	1.89
[C], $\frac{n}{p} = 512K$	4.13	4.02	16.4	14.9
[N], $\frac{n}{p} = 4K$	0.475	0.109	1.63	0.163
[N], $\frac{n}{p} = 64K$	0.907	0.613	3.55	1.89
[N], $\frac{n}{p} = 512K$	4.22	4.12	18.2	15.0

Input	CS-2 $p = 16$	
	[AIS+95]	[BHJ95]
[R], $\frac{n}{p} = 4K$	0.664	0.083
[R], $\frac{n}{p} = 64K$	1.33	0.808
[R], $\frac{n}{p} = 256K$	4.13	3.55
[R], $\frac{n}{p} = 512K$	7.75	7.33
[C], $\frac{n}{p} = 4K$	0.641	0.081
[C], $\frac{n}{p} = 64K$	1.23	0.790
[C], $\frac{n}{p} = 256K$	3.87	3.27
[C], $\frac{n}{p} = 512K$	6.86	6.65
[N], $\frac{n}{p} = 4K$	0.623	0.085
[N], $\frac{n}{p} = 64K$	1.22	0.815
[N], $\frac{n}{p} = 256K$	3.57	3.53
[N], $\frac{n}{p} = 512K$	6.34	7.29

Table I: Total Execution Time for Radix Sort on 32-bit Integers (in seconds), Comparing the AIS and Our Implementations

[8]Thanks to Mihai Ionescu and Klaus Schauser from UC Santa Barbara for providing the source code.

6 Acknowledgements

We would like to thank the CASTLE/Split-C group at The University of California, Berkeley, especially for the help and encouragement from David Culler, Arvind Krishnamurthy, and Lok Tin Liu. Computational support on UC Berkeley's 64-processor TMC CM-5 was provided by NSF Infrastructure Grant number CDA-8722788.

The University of California, Santa Barbara, parallel radix sort code was provided to us by Mihai Ionescu. Also, Klaus Schauser, Oscar Ibarra, Chris Scheiman, and David Probert of UC Santa Barbara, provided help and access to the UCSB 64-node Meiko CS-2. The Meiko CS-2 Computing Facility was acquired through NSF CISE Infrastructure Grant number CDA-9218202, with support from the College of Engineering and the UCSB Office of Research, for research in parallel computing.

Arvind Krishnamurthy provided additional help with his port of Split-C to the Cray Research T3D [3]. The Jet Propulsion Lab/Caltech 256-node Cray T3D Supercomputer used in this investigation was provided by funding from the NASA Offices of Mission to Planet Earth, Aeronautics, and Space Science. We also acknowledge William Carlson and Jesse Draper from the Center for Computing Science (formerly Supercomputing Research Center) for writing the parallel compiler AC (version 2.6) [15] on which the T3D port of Split-C has been based.

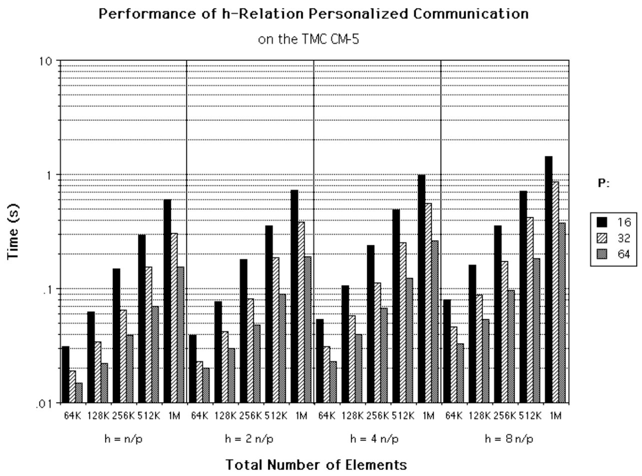
This work also utilized the CM-5 at National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, under grant number ASC960008N.

We also thank Jeffrey Hollingsworth from UMCP's Computer Science Department for his suggestions and encouragement.

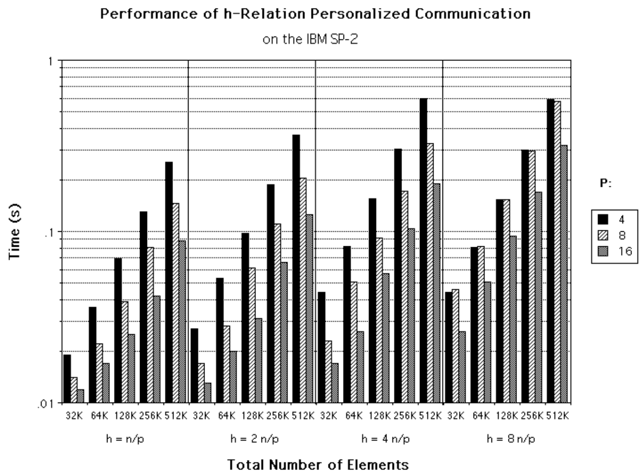
We would like to acknowledge the use of the UMIACS 16-node IBM SP-2-TN2, which was provided by an IBM Shared University Research award and an NSF Academic Research Infrastructure Grant No. CDA9401151.

Please see `http://www.umiacs.umd.edu/~dbader` for additional performance information. In addition, all the code used in this paper is freely available for interested parties from our anonymous ftp site, `ftp://ftp.umiacs.umd.edu/pub/dbader`. We encourage other researchers to compare with our results for similar inputs.

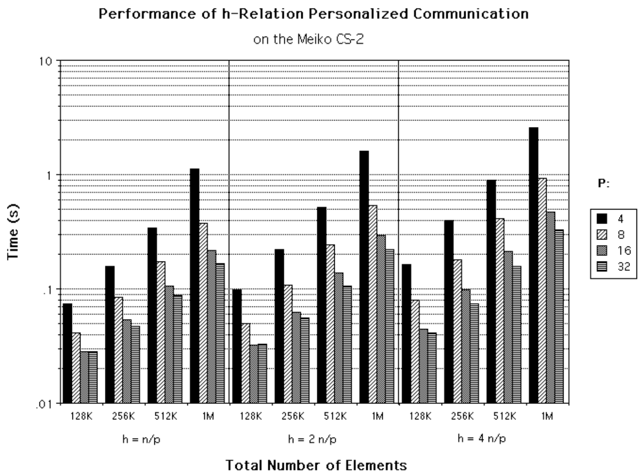
A Additional Performance Results



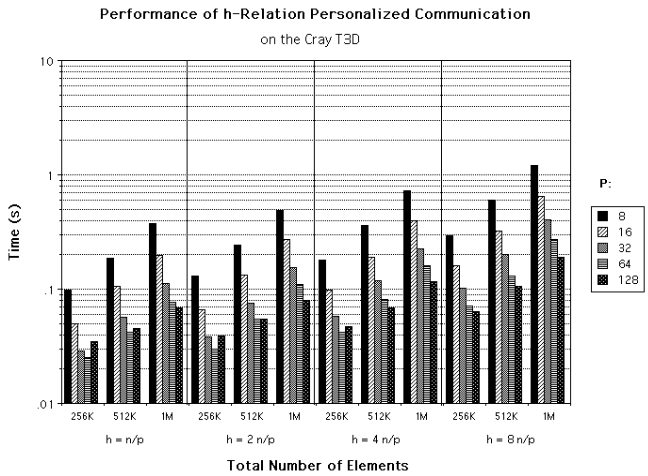
TMC CM-5



IBM SP-2



Meiko CS-2



Cray Research T3D

Figure 6: Performance of personalized communication with respect to machine and problem size

B Deterministic Routing Algorithm

The following is run on processor i:

Algorithm 1 *Deterministic Routing Algorithm*

Shared Memory Model Algorithm for routing an h-relation.

Input:

$\{ i \}$ is my processor number;
$\{ p \}$ is the total number of processors, labelled from 0 to $p-1$;
$\{ A \}$ is the $\frac{n}{p} \times p$ input array of elements (data, i);
$\{ B \}$ is the $h \times p$ output array;
$\{ T \}$ is the $p \times p$ array used for holding tags when placing elements;
$\{ C \}$ is an $\left(\frac{n}{p} + p^2 + p\right) \times p$ auxiliary array of elements (data, i);
$\{ D \}$ is an $\left(\frac{n}{p} + p^2 + p\right) \times p$ auxiliary array of elements (data, i);
$\{ E \}$ is an $(h + p^2) \times p$ auxiliary array of elements (data, i);
$\{ F \}$ is an $(h + p^2) \times p$ auxiliary array of elements (data, i);

begin

1. **Set** $\text{blksz1} = \frac{n}{p^2} + \frac{p}{2} + 1$.
2. **For** $k = 0$ **to** $p - 1$ **do**:
   2.1 **Set** $C[i][k * \text{blksz1}] = 0$.
   2.2 **Set** $T[i][k] = i + k \bmod p$.
3. **For** $k = 0$ **to** $\frac{n}{p} - 1$ **do**:
   3.1 **Set** $l = (A[i][k]) \rightarrow \text{address}$.
   3.2 **Set** $d = T[i][l]$.
   3.3 **Increment** $C[i][d * \text{blksz1}] \Longrightarrow m$.
   3.4 **Set** $C[i][d * \text{blksz1} + m] = A[i][k]$.
   3.5 **Set** $T[i][l] = T[i][l] + 1$ **mod** p.
4. $D =$ **transpose** (C).
5. **Set** $\text{blksz2} = \frac{h}{p} + \frac{p}{2}$.
6. **For** $k = 0$ **to** $p - 1$ **do**:
   6.1 **Set** $E[i][k * \text{blksz2}] = 0$.
7. **For** $k = 0$ **to** $p - 1$ **do**:
   7.1 **For** $l = 1$ **to** $D[i][k * \text{blksz1}]$ **do**:
      7.1.1 **Set** $d = (D[i][k * \text{blksz1} + l]) \rightarrow \text{address}$.
      7.1.2 **Increment** $E[i][d * \text{blksz2}] \Longrightarrow m$.
      7.1.3 **Set** $E[i][d * \text{blksz2} + m] = D[i][k * \text{blksz1} + l]$.
8. $F =$ **transpose**(E)
9. **Set** $z = 0$.
10. **For** $k = 0$ **to** $p - 1$ **do**:
    10.1 **For** $l = 1$ **to** $F[i][k * \text{blksz2}]$ **do**:
       10.1.1 **Set** $B[i][z] = F[i][k * \text{blksz2} + l]$.
       10.1.2 **Increment** z.

end

C Counting Sort Algorithm

Algorithm 2 *Counting Sort Algorithm*

Shared Memory Model Algorithm to sort n integer keys in the range $[0, R-1]$.

Input:

{ i } is my processor number;
{ p } is the total number of processors, labelled from 0 to $p-1$;
{ Key } is the $\frac{n}{p} \times p$ input array of integer keys in the range $[0, R-1]$;
{ $Addr$ } is the $\frac{n}{p} \times p$ array which is used for destination label of keys;
{ $Index$ } is the $R \times p$ array which is used for counting local keys;
{ $ScanTran$ } is the $\left(\frac{R}{p} \times p\right) \times p$ array which holds the transpose of $Index$;
{ $IntLeaveScan$ } is the $\left(2\frac{R}{p} \times p\right) \times p$ array which will be inverse transposed to $Scans$;
{ $Scans$ } is the $2R \times p$ array which is decomposed into $MyScan$ and $Total$;
{ $MyScan$ } is the $R \times p$ array which is used for holding the scan of Index;
{ $Total$ } is the $R \times p$ array which is used for holding the total count of keys;
{ *Offset* } is the $1 \times p$ array which is used for holding the current offset of rank;

begin

1. **For** $k = 0$ **to** $R-1$ **do**:
   1.1 **Set** $Index[i][k] = 0$.
2. **For** $k = 0$ **to** $\frac{n}{p} - 1$ **do**:
   2.1 **Increment** $Index[i]\,[Key[i][k]]$.
3. $ScanTran =$ **transpose**$(Index)$.
4. **For** $j = 0$ **to** $\frac{R}{p} - 1$ **do**:
   4.1 **For** $k = 1$ **to** $p-1$ **do**:
      4.1.1 **Set** $ScanTran[i]([k][j]) = ScanTran[i]([k-1][j]) + ScanTran[i]([k][j])$.

`/* Compose` *IntLeaveScan* `by interleaving scans in` *ScanTran*
`and totals in` *IntLeaveScan*`[p-1][⋆] */`

5. **For** $j = 0$ **to** $p-1$ **do**:
   5.1 **For** $k = 0$ **to** $\frac{R}{p} - 1$ **do**:
      5.1.1 **Set** $IntLeaveScan[i]([j][2k]) = ScanTran[i]([j][k])$.
      5.1.2 **Set** $IntLeaveScan[i]([j][2k+1]) = ScanTran[i]([p-1][k])$.
6. $Scans =$ **(inverse) transpose**$(IntLeaveScans)$.
7. **For** $k = 0$ **to** $R-1$ **do**: `/* Decompose` *Scans* `*/`
   7.1 **Set** $MyScan[i][k] = Scans[i][2k]$.
   7.2 **Set** $Total[i][k] = Scans[i][2k+1]$.
8. **Set** $Offset[i] = 0$.
9. **For** $k = 0$ **to** $R-1$ **do**:
   9.1 **Set** $Index[i][k] = MyScan[i][k] + Offset[i]$.
   9.2 **Set** $Offset[i] = Offset[i] + Total[i][k]$.
10. **For** $k = 0$ **to** $\frac{n}{p} - 1$ **do**:
    10.1 **Set** $Addr[i][k] = Index[i]\,[Key[i][k]]$.
    10.2 **Increment** $Index[i]\,[Key[i][k]]$.
11. **Routing of** h**-Relation** $(h = \frac{n}{p})$ of $\left\langle Key, \left(\textbf{proc: } \left\lfloor \frac{Addr}{p} \right\rfloor, \textbf{ position: } Addr \textbf{ mod } p\right)\right\rangle$.

end

References

[1] B. Abali, F. Özgüner, and A. Bataineh. Balanced Parallel Sort on Hypercube Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 4(5):572–581, 1993.

[2] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP model - One step closer towards a realistic model for parallel computation. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, Santa Barbara, CA, July 1995.

[3] R.H. Arpaci, D.E. Culler, A. Krishnamurthy, S.G. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In ACM Press, editor, *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 320–331, Santa Margherita Ligure, Italy, June 1995.

[4] D. Bader. Randomized and Deterministic Routing Algorithms for h-Relations. ENEE 648X Class Report, April 1, 1994.

[5] D. A. Bader and J. JáJá. Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study. Technical Report CS-TR-3384 and UMIACS-TR-94-133, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, December 1994.

[6] D. A. Bader and J. JáJá. Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study. In *Fifth ACM SIGPLAN Symposium of Principles and Practice of Parallel Programming*, pages 123–133, Santa Barbara, CA, July 1995.

[7] D. A. Bader and J. JáJá. Practical Parallel Algorithms for Dynamic Data Redistribution, Median Finding, and Selection. Technical Report CS-TR-3494 and UMIACS-TR-95-74, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, July 1995.

[8] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Technical Report RNR-94-007, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Moffett Field, CA, March 1994.

[9] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.-T. Ho, S. Kipnis, and M. Snir. CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 6:154–164, 1995.

[10] D.P. Bertsekas, C. Özveren, G.D. Stamoulis, P. Tseng, and J.N. Tsitsiklis. Optimal Communication Algorithms for Hypercubes. *Journal of Parallel and Distributed Computing*, 11:263–275, 1991.

[11] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

[12] S.H. Bokhari. Complete Exchange on the iPSC-860. ICASE Report No. 91-4, ICASE, NASA Langley Research Center, Hampton, VA, January 1991.

[13] S.H. Bokhari. Multiphase Complete Exchange on a Circuit Switched Hypercube. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages I–525 – I–529, August 1991. Also appeared as NASA ICASE Report No. 91-5.

[14] S.H. Bokhari and H. Berryman. Complete Exchange on a Circuit Switched Mesh. In *Proceedings of Scalable High Performance Computing Conference*, pages 300–306, Williamsburg, VA, April 1992.

[15] W.W. Carlson and J.M. Draper. AC for the T3D. Technical Report SRC-TR-95-141, Supercomputing Research Center, Bowie, MD, February 1995.

[16] Cray Research, Inc. *SHMEM Technical Note for C*, October 1994. Revision 2.3.

[17] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, S. Luna, T. von Eicken, and K. Yelick. *Introduction to Split-C.* Computer Science Division - EECS, University of California, Berkeley, version 1.0 edition, March 6, 1994.

[18] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[19] V.V. Dimakopoulos and N.J. Dimopoulos. Optimal Total Exchange in Linear Arrays and Rings. In *Proceedings of the 1994 International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 230–237, Kanazawa, Japan, December 1994.

[20] A.C. Dusseau. Modeling Parallel Sorts with LogP on the CM-5. Technical Report UCB//CSD-94-829, Computer Science Division, University of California, Berkeley, 1994.

[21] N. Folwell, S. Guha, and I. Suzuki. A Practical Algorithm for Integer Sorting on a Mesh-Connected Computer. In *Proceedings of the High Performance Computing Symposium*, pages 281–291, Montreal, Canada, July 1995. Preliminary Version.

[22] A.V. Gerbessiotis and L.G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

[23] S. Heller. Congestion-Free Routing on the CM-5 Data Router. In *Proceedings of the First International Workshop on Parallel Computer Routing and Communication*, pages 176–184, Seattle, WA, May 1994. Springer-Verlag.

[24] S. Hinrichs, C. Kosak, D.R. O'Hallaron, T.M. Strickler, and R. Take. An architecture for optimal all-to-all personalized communication. Technical Report CMU-CS-94-140, School of Computer Science, Carnegie Mellon University, September 1994.

[25] T. Horie and K. Hayashi. All-to-All Personalized Communication on a Wrap-around Mesh. In *Proceedings of the Second Fujitsu-ANU CAP Workshop*, Canberra, Austrailia, November 1991. 10 pp.

[26] J. JáJá and K.W. Ryu. The Block Distributed Memory Model. Technical Report CS-TR-3207, Computer Science Department, University of Maryland, College Park, January 1994.

[27] J.F. JáJá and K.W. Ryu. The Block Distributed Memory Model for Shared Memory Multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 752–756, Cancún, Mexico, April 1994. (Extended Abstract).

[28] S.L. Johnsson and C.-T. Ho. Optimal Broadcasting and Personalized Communication in Hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, 1989.

[29] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley Publishing Company, Reading, MA, 1973.

[30] D. Krizanc. Integer Sorting on a Mesh-Connected Array of Processors. *Information Processing Letters*, 47(6):283–289, 1993.

[31] Y.-D. Lyuu and E. Schenfeld. Total Exchange on a Reconfigurable Parallel Architecture. In *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, pages 2–10, Dallas, TX, December 1993.

[32] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, June 1995. Version 1.1.

[33] S.R. Öhring and S.K. Das. Efficient Communication in the Foldned Petersen Interconnection Networks. In *Proceedings of the Sixth International Parallel Architectures and Languages Europe Conference*, pages 25–36, Athens, Greece, July 1994. Springer-Verlag.

[34] S. Ranka, R.V. Shankar, and K.A. Alsabti. Many-to-many Personalized Communication with Bounded Traffic. In *The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 20–27, McLean, VA, February 1995.

[35] S. Rao, T. Suel, T. Tsantilas, and M. Goudreau. Efficient Communication Using Total-Exchange. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 544–550, Santa Barbara, CA, April 1995.

[36] T. Schmiermund and S.R. Seidel. A Communication Model for the Intel iPSC/2. Technical Report Technical Report CS-TR 9002, Dept. of Computer Science, Michigan Tech. Univ., April 1990.

[37] D.S. Scott. Efficient All-to-All Communication Patterns in Hypercube and Mesh Topologies. In *Proceedings of the 6th Distributed Memory Computing Conference*, pages 398–403, Portland, OR, April 1991.

[38] R.V. Shankar, K.A. Alsabti, and S. Ranka. The Transportation Primitive. Personal communication, August 1994.

[39] T. Suel. Routing and Sorting on Meshes with Row and Column Buses. Technical Report UTA//CS-TR-94-09, Department of Computer Sciences, University of Texas at Austin, October 1994.

[40] R. Take. A Routing Method for All-to-All Burst on Hypercube Networks. In *Proceedings of the 35th National Conference of Information Processing Society of Japan*, pages 151–152, 1987. In Japanese. Translation by personal communication with R. Take.

[41] R. Thakur and A. Choudhary. All-to-All Communication on Meshes with Wormhole Routing. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 561–565, Cancún, Mexico, April 1994.

[42] R. Thakur, A. Choudhary, and G. Fox. Complete Exchange on a Wormhole Routed Mesh. Report SCCS-505, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY, July 1993.

[43] R. Thakur, R. Ponnusamy, A. Choudhary, and G. Fox. Complete Exchange on the CM-5 and Touchstone Delta. *Journal of Supercomputing*, 8:305–328, 1995. (An earlier version of this paper was presented at Supercomputing '92.).

[44] L.G. Valiant. A Bridging Model for Parallel Computation. *Communication of the ACM*, 33(8):103–111, 1990.

[45] J.-C. Wang, T.-H. Lin, and S. Ranka. Distributed Scheduling of Unstructured Collective Communication on the CM-5. Technical Report CRPC-TR94502, Syracuse University, Syracuse, NY, 1994.

[46] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.