Fast Parallel Sorting under LogP: Experience with the CM-5

Andrea C. Dusseau, David E. Culler, Klaus Erik Schauser, Richard P. Martin
Computer Science Division
University of California, Berkeley
{dusseau, culler, schauser, rmartin}@CS.Berkeley.EDU

Abstract

In this paper, the LogP model is used to analyze four parallel sorting algorithms (bitonic, column, radix, and sample sort). LogP characterizes the performance of modern parallel machines with a small set of parameters: the communication latency (L), overhead (o), bandwidth (g), and the number of processors (P). We develop implementations of these algorithms in Split-C, a parallel extension to C, and compare the performance predicted by LogP to actual performance on a CM-5 of 32 to 512 processors for a range of problem sizes and input sets. The sensitivity of the algorithms is evaluated by varying the distribution of key values and the rank ordering of the input.

The LogP model is shown to be a valuable guide in the development of parallel algorithms and a good predictor of implementation performance. The model encourages the use of data layouts which minimize communication and balanced communication schedules which avoid contention. Using an empirical model of local processor performance, LogP predictions closely match observed execution times on uniformly distributed keys across a broad range of problem and machine sizes for all four algorithms. Communication performance is oblivious to the distribution of the keys values, whereas the local sort performance is not. The communication phases in radix and sample sort are sensitive to the ordering of keys, because certain layouts result in contention.

1 Introduction

Sorting is important in a wide variety of practical applications, is interesting to study from a theoretical viewpoint, and offers a wealth of novel parallel solutions. The richness of this particular problem arises, in part, because it fundamentally requires communication as well as computation. Thus, sorting is an excellent area in which to investigate the translation from theory to practice of novel parallel algorithms on large parallel systems.

Parallel sorting algorithms have generally been studied either in the context of PRAM-based models, with uniform access to the entire data set, or in network-based models, with communication allowed only between neighbors in a particular interconnection topology. In both approaches, algorithms are typically developed under the assumption that the number of processors (P) is comparable to the number of data elements (N), and then an efficient simulation of the algorithm is provided for the case where $P < N$.

In this paper, we study fast parallel sorting from the perspective of a new "realistic" parallel model, LogP[1], which captures the key performance characteristics of modern large scale multiprocessors, such as the Thinking Machines CM-5. In particular, the model reflects the technological reality that these machines are essentially a collection of workstation-class nodes which communicate by point-to-point messages that travel through a dedicated, high performance network. The LogP model downplays the role of the topology of the interconnection network and instead describes its performance characteristics.

Because the individual processors of today's multiprocessors have substantial computing power and a substantial amount of memory, the most interesting problems, and the problems on which the machine performs best, have many data elements per processor. One of the interesting facets of parallel sorting algorithms is how they exploit this grouping of data within processors. In this context, fast sorting algorithms tend to have three components: a purely local computational phase which exploits the grouping of elements onto processors, an intermediate phase which determines the specific transformation, and a communication phase which often involves a general transformation of the entire data set.

Our implementation language, Split-C [2], provides an attractive basis for this study, because it exposes the capabilities modeled by LogP through a rich set of assignment operators in a distributed global address space. Traditional shared memory models would force us to use only read/write as the access primitives; traditional message passing models would impose some variant of send and receive, with its associated protocol overhead; and data parallel languages would place a complex compiler transformation between the written program and the actual executable. Split-C, like C, provides a straightforward machine independent programming system, without attempting to hide the underlying performance characteristics of the machine.

We were strongly influenced in this study by a previous comparison of sorting algorithms, which examined bitonic, radix, and sample sort implemented in microcode on the CM-2[3]. We augment the comparison to include column sort, address a more general class of machines, formalized by LogP, and implement the algorithms in a language that can be ported to a variety of parallel machines.

This paper is organized as follows. In Section 2, the LogP model is described. In Section 3, we describe our experimental environment, consisting of our implementation language, Split-C, the input data set used in our measurements, and the LogP characterization of the CM-5, as well as our model for the local processors. In the next four sections, we examine four sorting algorithms: bitonic sort[4], column sort[5], radix sort[3, 6], and sample sort[3]. The order in which we discuss the sorts is based on the increasing complexity of their communication phases. We predict the execution time of each algorithm based on the four parameters of the LogP model and a small set of parameters that characterize the computational performance of the individual processing nodes, such as the time for a local sort. When discussing bitonic sort and radix sort, special attention is paid to two of the most interesting communication phases: a remap and a multi-scan, respectively. The predictions of the model are compared to measurements of the algorithms on the CM-5 for a variety of input sets. Finally, in Section 8, we compare the performance of the four algorithms.

2 LogP

The LogP model[1] reflects the convergence of parallel machines towards systems formed by a collection of complete computers, each consisting of a powerful microprocessor, cache, and large DRAM memory, connected by a communication network.

Since there appears to be no consensus emerging on interconnection topology — the networks of new commercial machines are typically different from their predecessors and different from each other — attempting to exploit a specific network topology is likely to yield algorithms that are not very portable. LogP avoids specifying the structure of the network and instead recognizes three parameters of the network. First, inter-processor communication involves a large delay, as compared to a local memory access. Secondly, networks have limited bandwidth per processor, as compared to the local memory or local cache bandwidth. This bandwidth may be further reduced by contention for the destination. Thirdly, there is a cost to the processors involved at both ends of a communication event; this cost is independent of the transmission latency between processors.

Specifically, LogP is a model of a distributed-memory multiprocessor in which processors communicate through point-to-point messages and whose performance is characterized by the following parameters.

L: an upper bound on the *latency*, or delay, incurred in communicating a message containing a small, fixed number of words from its source processor/memory module to its target.

o: the *overhead*, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor cannot perform other operations.

g: the *gap*, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g is the available per-processor communication bandwidth.

P: the number of processor/memory modules. The characteristics of the processor are not specified by the model

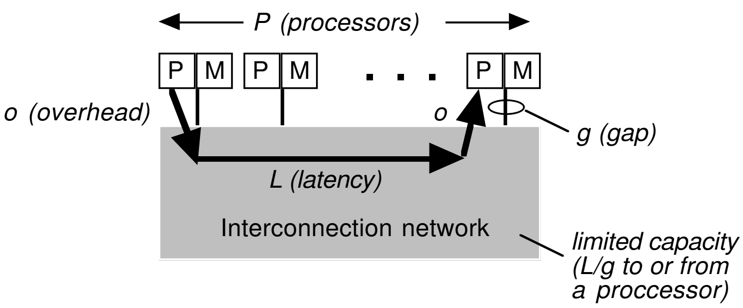

Figure 1: *The LogP model describes an abstract machine configuration in terms of four performance parameters: L, the latency experienced in each communication event, o, the overhead experienced by the sending and receiving processors for each communication event, g, the gap between successive sends or successive receives by a processor, and P, the number of processors/memory modules.*

L, o, and g are specified in units of time. As will become clear in analyzing the sorting algorithms, the parameters are not equally important in all situations; often it is possible to ignore one or more parameters and work with a simpler model. The model assumes that all messages are of a “small size”, which we call the communication word and denote by w.

The model is *asynchronous*, so processors work asynchronously and the latency experienced by any message is unpredictable, but is bounded above by L in the absence of stalls. In estimating the running time of an algorithm, we assume that each message incurs a latency of L. Furthermore, it is assumed that the network has a *finite capacity*, such that at most $\lceil L/g \rceil$ messages can be in transit from any processor or to any processor at any time. If a processor attempts to transmit a message that would exceed this limit, it stalls until the message can be sent without exceeding the capacity limit. No position is taken on how a processor is notified of the arrival of a message, *e.g.*, through an interrupt or by polling. However, if a processor ignores the arrival of a message for some time, sending processors could stall as a result of exceeding the capacity limit.

By charging for communication events, the model favors algorithms that minimize communication, *e.g.*, by exploiting the grouping of a large number of data elements on a processor. Where communication does occur, its latency can be masked by simultaneous use of the processor. Although the model does not explicitly specify the characteristics of the local processor, it assumes that optimizing local processor performance is important. Our model of the local processor is presented in Section 3.3.

2.1 Use of the model

To demonstrate the use of the model, let us consider some simple inter-processor operations. As illustrated in Figure 1, the simplest operation is the transfer of a communication word from one processor to another. This operation requires a total time of $L + 2o$ and each processor is busy for o units of time. A remote read operation involves two such messages, so it has a total time of $2L + 4o$, where the processor issuing the read and the one servicing the read each spend time $2o$ interacting with the network.[1]

The more interesting case to consider is a sequence of messages; this illustrates pipelining of communication and the role of the bandwidth limit. We shall assume throughout that $o < g$, since otherwise a processor cannot utilize the network bandwidth and g can be ignored. If one processor sends n messages to another the total time is $2o + (n-1)g + L$. The sending processor spends o units of time delivering the first message into the network. It can then deliver each additional message at an interval of time g. These each take time L to reach the destination, and then the destination processor is busy for time o after receiving each. If n is large, then the $2o$ and L terms can be ignored.

The timing analysis is identical if one processor transfers n communication words to many others, except that the receive overhead is distributed across the receivers; the sender still issues messages at a rate of g. In the case where many processors are sending to a single processor, the total time for the operation is the same, because the time is limited by the receive bandwidth of the destination. However, the cost to the senders is greater, since processors stall as a result of exceeding the capacity constraint.

If pairs of processors exchange n messages each, then the analysis is more complicated. Assuming that both processors begin sending at the same time, for L units of time each processor sends messages at an interval of g. After time L, each processor both sends and receives messages, so their sending rate slows to $\max(g, 2o)$. After all messages have been sent, each processor receives messages at the sending rate.

[1]On machines with hardware for shared memory access, the remote end may be serviced by an auxiliary processor that is part of the memory controller[7].

Therefore, the total time for this operation is $2L + 2o + (n - 1 - L/g) \max(g, 2o)$. Note that this assumes the processors alternate between sending and receiving after the first message arrives. If n is large this equation can be approximated with $n * \max(g, 2o)$.

In many cases, additional performance is achieved if w words are transferred in a single message. For example, the time for pairs of processors to exchange n computational words with $\lceil n/w \rceil$ messages is

$$T_{\text{exch}}(n) = 2L + 2o + (\lceil n/w \rceil - 1 - L/g) \max(g, 2o).$$

The formula for T_{exch} is used in several of the sorting algorithms.

3 Experimental Environment

In this section, we present our experimental environment. We begin by describing the relevant features of our implementation language, Split-C. We then discuss the probability distribution of the input keys used in our measurements. Next, we characterize the CM-5 in terms of the LogP parameters. Finally, we discuss our model of the local computation, focusing on the local sort.

3.1 Split-C

Our sorting algorithms are written in Split-C[2], a parallel extension of the C programming language that can express the capabilities offered by the LogP model. The language follows a SPMD (single program multiple data) model. Processors are distinguished by the value of the special constant, `MYPROC`. Split-C provides a shared global address space, comprised of the address space local to each processor. Programs can be optimized to take advantage of the grouping of data onto processors by specifying the data layout with *spread arrays*. Two forms of pointers are provided in Split-C: standard pointers refer to the region of the address space local to the referencing processor, *global pointers* refer to an address anywhere in the machine. The time to read or write the data referenced by a global pointer under LogP is $2L + 4o$, since a request is issued and a response returned. For the read, the response is the data, for the write it is the completion acknowledgement required for consistency.

The unusual aspects of Split-C are the assignment operations that allow the programmer to overlap communication and avoid unnecessary communication events. With *split-phase* assignment, expressed with `:=`, the initiating processor does not wait for a response, so $2L + 2o$ cycles of useful work can be performed during the remote operation. In particular, the processor can initiate additional communication requests. With a *signalling store*, expressed with `:-`, an acknowledgement is not returned to the initiating processor, so the operation only requires time $L + 2o$. Bulk transfer of multiple words is provided for all of the described styles of communication.

3.2 Input Key Characterization

In this study, we focus on the expected performance of the algorithms when sorting random, uniformly-distributed 31-bit keys.[2] We compare the predicted time per key for the four algorithms to the measured time per key with this distribution of keys on 32 through 512 processors and for 16K to 1M keys per processor. Throughout the paper, N designates the total number of keys to be sorted, where each processor initially has $n = N/P$ keys.

We evaluate the robustness of the algorithms to variations in the input set by measuring their performance on non-uniform data sets. This analysis is performed on a fixed number of processors (64) and a fixed number of keys per processor (1M). For the sorting algorithms whose communication phases are oblivious to the values of the input keys (*i.e.*, bitonic and column sort), we do not evaluate the effect of the layout of keys across processors on the performance; we only look at the effect of input sets with different probability distributions.

The probability distribution of each input set is characterized by its Shannon entropy[8], defined as

$$-\sum_{i=1}^{N} p_i * \lg p_i,$$

where p_i is the probability associated with key i. To generate input data sets with various entropies, we produce keys whose individual bits have between 0 and 1 bits of entropy. Multiple keys from a uniform distribution are combined into a single key having a non-uniform distribution, as suggested in [9]. For example, if the binary AND operator is applied to two independent keys generated from a uniform distribution, then each bit in the resulting key has a 0.75 chance of being a zero and a 0.25 chance of being a one. This produces an entropy of 0.811 for each bit; for 31-bit keys the resulting entropy is 25.1. By AND'ing together more keys we produce input sets with lower entropy. Our test suite consists of input sets with entropy 31, 25.1, 16.9, 10.4, 6.2, and 0 (a constant value) randomly distributed across processors.

For the sorting algorithms whose communication phases are dependent upon the values of the keys (*i.e.*, radix and sample sort), we also evaluate how the initial layout of the keys across processors affects performance (*e.g.*, keys sorted into a blocked layout versus keys sorted cyclically). We determine the best and worst case initial layouts of keys, which are the layouts which produce, respectively, either no communication between processors or the greatest amount of contention during communication.

3.3 CM-5 LogP Characterization

Our measurements are performed on the Thinking Machines CM-5, a massively parallel MIMD computer based on the Sparc processor. Each node consists of a 33 MHz Sparc RISC processor chip-set and a network interface. The nodes are interconnected in two identical disjoint incomplete fat trees, and a broadcast/scan/prefix control network. The implementations of the sorting algorithms do not use the vector accelerators.

In previous experiments on the CM-5[10, 1], we determined that $o \approx 2.2\mu s$ and, on an unloaded

[2] Our random number generator produces numbers in the range 0 through $2^{31} - 1$.

network, $L \approx 6\mu s$. The communication word size, w, is equal to four (32-bit) processor words. The bisection bandwidth[3] is 5 MBytes/s per processor, so we take g to be $4\mu s$.

3.4 CM-5 Processor Characterization

LogP does not specify how the local processor is to be modeled. Modeling local computation is not the focus of this study, yet is necessary in order to predict the total execution time of the algorithms. Therefore, we characterize the local performance empirically. We assign a time per key per processor for each of the local computation phases, such as merging two sorted lists (t_{merge}) or clearing histogram buckets (t_{zero}). Throughout our analysis, we use a lowercase t to indicate a rate per key and an uppercase T for the total time of a phase.

The local sort plays an important role in bitonic, column and sample sort, accounting for up to 80% of the execution time. Thus, it is important both to optimize the local sort carefully and to model its performance accurately. We determined empirically, for the number of keys per processor in this study and for uniformly distributed keys, that an 11-bit radix sort is faster than radix sorts of other digit sizes and quicksort. Radix sort relies on the representation of keys as b-bit numbers. Each key is divided into $\lceil b/r \rceil$ digits of r bits each, where r is the *radix*. One pass is then performed over each digit, each time permuting the keys according to the rank of the digit.

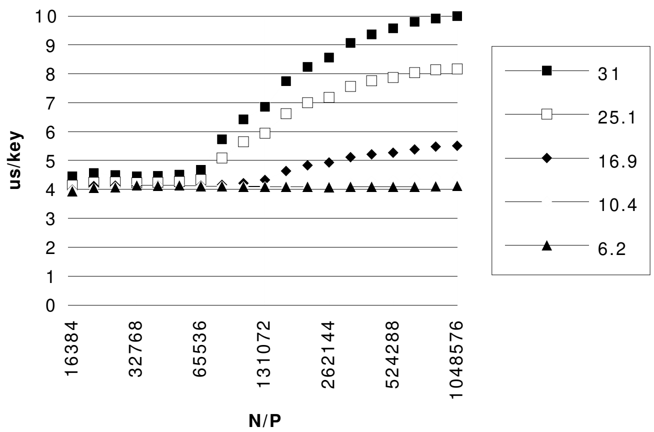

Figure 2: *Measured execution times per key for the local radix sort for input key distributions with entropies of 31, 25.1, 16.9, 10.4, and 6.2.*

Figure 2 shows the measured time per key of the local sort as n is varied from 16 K to 1 M keys per processor and for keys with entropies between 31 and 6.2. As evident from the figure, the execution time when sorting uniform keys is very sensitive to the number of keys per processor; however, when sorting keys with a lower entropy, the performance varies much less with the number of keys. Simulations of the local radix sort and the memory hierarchy of the CM-5 nodes reveal that the variation in execution time is largely due to TLB misses that occur when the keys are written into their ordered positions on each pass.

[3] The bisection bandwidth is the minimum bandwidth through any cut of the network that separates the set of processors into halves.

With a uniformly distributed input set, on each of the $\lceil b/r \rceil$ passes, the destination position of each key is essentially random and, thus, is written to a random page of the destination buffer. Assuming a page of memory holds k keys, then n/k pages and TLB entries are required for the destination buffer. If l otherwise inactive TLB entries exist for the destination array,[4] then the probability that a key's destination page is not contained in the TLB is $(1 - l * k/n)$. If $t_{\text{localsort\_tlbhit}}$ is the time per key for the local radix sort when all destination pages are contained in the TLB, we can model the time of the local sort on uniformly distributed data as follows.

$$t_{\text{localsort}} = \begin{cases} t_{\text{localsort\_tlbhit}} & \text{if } n \leq (l * k) \\ t_{\text{localsort\_tlbhit}} + \left\lceil \frac{b}{r} \right\rceil t_{\text{tlb\_miss}} * (1 - \frac{l*k}{n}) & \text{otherwise} \end{cases}$$

On the CM-5, $t_{\text{localsort\_tlbhit}}$ is measured as $4.5\mu s$/key, $t_{\text{tlb\_miss}}$, the cost of replacing an entry in TLB, is approximately $1.5\mu s$/key, l is $(64 - 3 = 61)$ TLB entries, and k is 1K keys. Substituting these numbers into our model gives the formula for $t_{\text{localsort}}$in Table 1.

The model for the local sort is within 10% of the measured time of the local sort on uniform keys; however, the performance of the local radix sort also depends upon the probability distribution of the input keys. Since identical digits have adjacent positions in the destination buffer, sorting many keys with the same value increases the likelihood that the same digit, and thus the same destination page, is referenced. If few unique keys exist, then the probability of hitting in the TLB increases and the local sort time decreases. For example, when sorting identical keys, TLB misses occur only when a page boundary is crossed. Rather than developing a model which predicts the probability of missing in the TLB as a function of the input key distribution, we use the model for uniformly distributed keys as an upper bound on the execution time.

The other computation steps account for a much smaller fraction of the execution time of the algorithms. For this reason, we use a measured time per key per processor for the computation rates. When possible, we use a constant rate for the time per key; however, in the case of T_{swap}, T_{gather}, and T_{scatter}, we use times per key that are dependent upon the number of keys, because the computation is sensitive to memory effects. Table 1 shows the local computational rates used in all of the sorts.

4 Bitonic Sort

In this section, we discuss a variant of Batcher's bitonic sort[4]. After describing the general algorithm, we present a data layout that reduces communication and enables optimizations for the local computation. We then describe how the LogP model guides us to an efficient implementation of the important communication operations: remaps between cyclic and blocked layouts. Finally, we give the predicted execution time under LogP and compare it to measured results.

Bitonic sort is based on repeatedly merging two *bitonic sequences* to form a larger bitonic sequence.[5] The basic algorithm for sorting N numbers performs $\lg N$ *merge stages*. The communication structure

[4] We assume that the source buffer of keys, the code, and the histogram buckets each require one active TLB entry.

[5] A bitonic sequence is a sequence that can be circularly shifted such that it first increases monotonically and then decreases monotonically.

Variable	Operation	Time Per Key (μs/key)		Sort
t_{swap}	simulate cyclic butterfly for key	$-0.08 + 0.025 * \lg n$		Bitonic
$t_{\text{mergesort}}$	sort bitonic sequence of keys	1.0		
t_{scatter}	move key for cyclic to blocked remap	0.46 $0.44 + 0.00059 * P$	if $n \leq$ 64K otherwise	
t_{gather}	move key for blocked to cyclic remap	0.52 1.1	if $n \leq$ 64K or $P \leq 64$ otherwise	Bitonic and Column
$t_{\text{localsort}}$	local radix sort of random keys	4.5 $9.0 - (281088/n)$	if $n <$ 64K if 64K $\leq n$	
t_{merge}	merge two sorted lists	1.5		Column
$t_{\text{shiftcopy}}$	shift key	0.5		
t_{zero}	clear histogram bin	0.22		Radix
t_{h}	produce histogram	1.2		
t_{add}	produce scan value	1.0		
t_{bsum}	adjust scan of bins	2.5		
t_{addr}	determine destination	4.7		
t_{compare}	compare key to splitter	0.9		Sample
$t_{\text{localsort\_8}}$	local radix sort of samples	5.0		

Table 1: *Models of local computation rates.*

of the i-th merge stage can be represented by $N/2^i$ butterflies each with 2^i rows and i columns. Each butterfly node compares two keys and selects either the maximum or the minimum key. The communication structure of the complete algorithm can be visualized as the concatenation of increasingly larger butterflies, as suggested by Figure 3.

The basic algorithm does not specify the layout of keys across processors nor what operations are performed by each processor. The standard approach to implementing bitonic sort is to simulate the individual steps in the butterfly. However, we derive a more efficient data placement that was inspired by the mapping used for large FFTs [1].

Our bitonic sort starts with a blocked layout. The first n keys are assigned to the first processor, which is responsible for the operations represented by the first n rows of the butterfly nodes, the second n keys and n rows are assigned to the second processor, and so on. Under this layout, the first $\lg n$ merge stages are entirely local. Since the purpose of these first stages is to form a monotonically increasing or decreasing sequence of n keys on each processor, we can replace all of these merge stages with a single, highly optimized *local sort*. For example, with two processors the first two merge stages in Figure 3 are entirely local and are replaced with a local sort.

For subsequent merge stages, we remap from a blocked to a cyclic layout. Under a cyclic layout, the

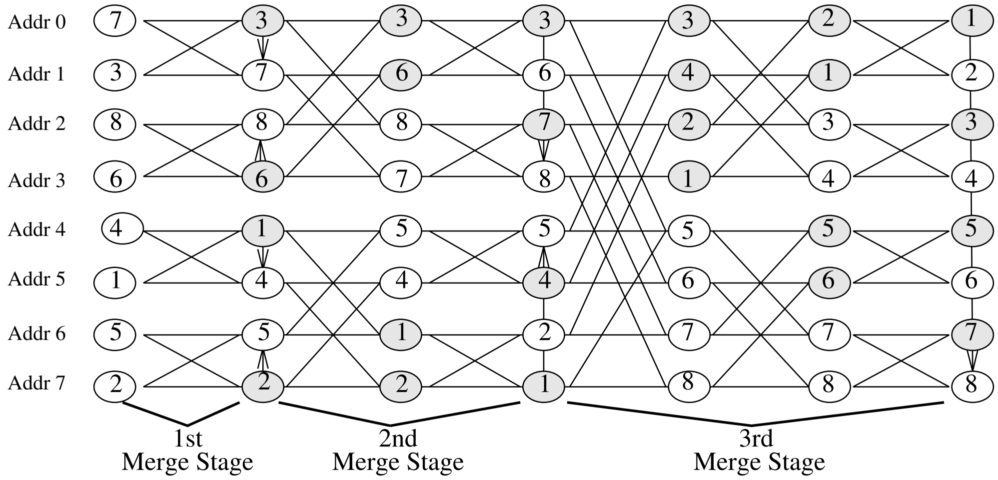

Figure 3: *The communication required for bitonic sort to sort eight keys using a series of merge stages. A row of nodes represents an address containing one of the keys. The edges show which keys are swapped with one another. A shaded node designates an address where the minimum of the two keys is placed. The arrows indicate the monotonic ordered sequences, with the arrowhead pointing towards the largest key.*

first key is assigned to the first processor, the second key to the second processor, and so on. The first $i - \lg n$ columns of the i-th merge stage are computed locally. In each of these steps, the processor performs a comparison and conditional swap of pairs of keys. A remap back into a blocked layout is then performed so the last $\lg n$ steps of the merge stage are local.

A gather or scatter operation is associated with each remap, but by restructuring the local computation on either side of the remap, it is often possible to eliminate the gather and/or the scatter. The purpose of the final $\lg n$ steps of each stage is again to produce sorted sequences of n keys on every processor. Since at this point the keys on each processor form a bitonic sequence, we can use a *bitonic merge sort*, rather than full radix sort. In a bitonic merge sort, after the minimum element has been found, the keys to its left and right are simply merged.

4.1 Optimizing Remaps

The remaps between cyclic and blocked layouts involve *regular* and *balanced* all-to-all communication, *i.e.*, the communication schedule is oblivious to the values of the keys and each processor receives as much data as it sends. The remap operation consists of $(P - 1)$ iterations, where on each iteration the processor performs a gather or scatter and exchanges n/P keys with another processor. Recall that one source of stalls under LogP occurs when the capacity limit of the network is exceeded. We can construct a communication schedule that ensures that no more than L/g messages are in transit to any processor by having processor p exchange data with processor $p \oplus i$ on iteration i.

Ignoring the gather and scatter cost, the remap is modeled as $(P - 1)$ iterations of pair-wise bulk exchanges of n/P keys, where the time for a single exchange was presented in Section 2.[6]

[6]The astute reader may notice that there appears to be an opportunity to save time $(P - 2)L$ by not waiting for the pairwise exchange to complete before storing into the next processor. However, given that processors operate asynchronously, due to cache

$$T_{\text{remap}} = (P-1) * T_{\text{exch}}(n/P)$$

Stalls may also occur under LogP if a processor fails to retrieve messages from the network. In Split-C, the network is polled when messages are sent or when explicitly specified. Therefore, if local computation, such as a gather or scatter, is performed during each iteration of the remap, messages may not be retrieved from the network after time L as expected. The simple solution is to remove the local computation from the communication loop, *i.e.*, gather the data for all processors before storing to any of them. With this approach, the execution time of remap is within 15% of that predicted by the model for all values of n and P.[7] We would like to point out that our first attempts to implement the remap ignored the network and caused contention; thus, we failed to completely address the potential stalls articulated in the model and the performance suffered. Not meeting the model predictions motivated us to look more closely at the implementation, and the model served further to explain the defect.

4.2 LogP Complexity

In this section, we summarize the LogP complexity of bitonic sort. Our bitonic sort algorithm begins with a blocked layout and performs a local sort. Next, lg P merge stages are performed. Each merge stage consists of a remap from a blocked to a cyclic layout, a sequence of local swaps, a remap of the data back to a blocked layout, and a bitonic merge sort.

$$\begin{aligned} T_{\text{bitonic}} &= n * t_{\text{localsort}} + n * t_{\text{gather}} \\ &\quad + \lg P * (T_{\text{remap}} + \frac{1}{2}(\lg P + 1) * n * t_{\text{swap}} + T_{\text{remap}} + n * t_{\text{scatter}} + n * t_{\text{mergesort}}) \end{aligned}$$

The equation for the communication phase, T_{remap}, was given in Section 4.1.

4.3 Empirical Results

Figure 4 shows the predicted and measured time per key for bitonic sort on 16K through 1M keys per processor on a CM-5 of 32 to 512 processors. Each data point represents a single run of the program. These experiments were performed on random distributions of keys, *i.e.*, their entropy is equal to 31. The time per key per processor increases only slightly with problem size, but increases substantially with the number of processors. The increase in time per key across processors is largely due to the greater number of merge stages. Comparing the two figures, it is evident that our prediction of the overall execution time per key is close to the measured time; all of our errors are less than 12%.

Figure 5 shows the breakdown of the execution time of bitonic sort into computation and communication phases. Predicted and measured times are presented for 512 processors. The figure illustrates that the time per key increases slowly with n due to the local steps: the local sort, the swap, and the bitonic merge sort.

misses and network collisions, this approach increases the likelihood that there will be contention.

[7] The model used in this comparison and in the graphs to follow was adjusted to include the overhead of Split-C's implementation of bulk stores. An additional cost of $2L + 4o$ is incurred for each invocation of bulk store to set up a communication segment.

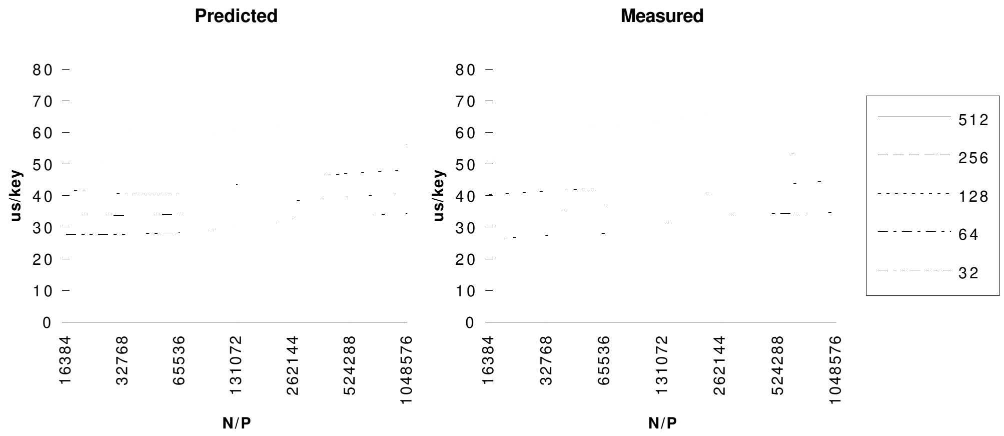

Figure 4: *Predicted and measured execution time per key of bitonic sort on the CM-5. Times are shown to sort between 16K and 1M keys per processor on 32, 64, 128, 256 and 512 processors.*

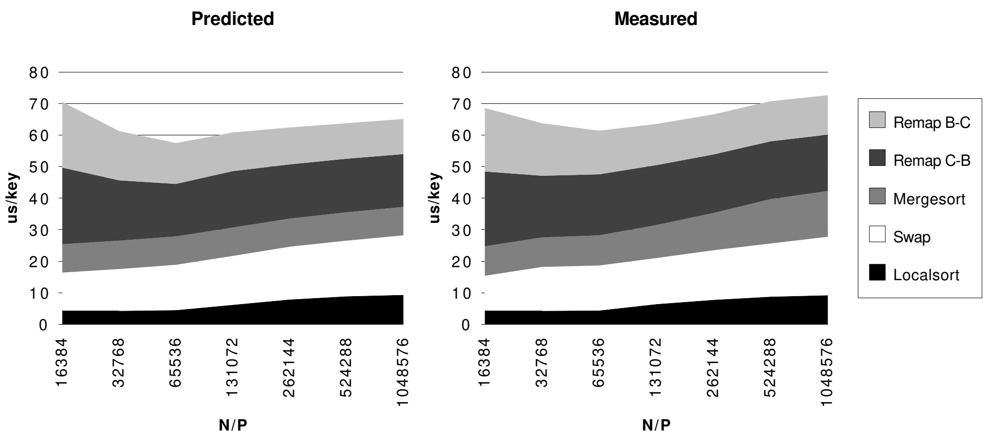

Figure 5: *Predicted and measured execution times per key on 512 processors for the phases of bitonic sort. The time for the single gather is included in the time for the remap from a blocked to a cyclic layout; likewise, the time for the scatter is included in the time for the remap from a cyclic to a blocked layout.*

The increase in each of these steps is due to cache effects. The time per key for the communication steps actually decreases slightly with n because the remap contains a startup cost proportional to the number of processors. These two trends imply that the percentage of time spent performing communication decreases with n, from 64% at small data sets to 40% at large data sets.

The communication operations performed in bitonic sort are oblivious to the distribution of the input set, and, as our experiments demonstrate, the total execution time of bitonic sort is relatively insensitive to the input set. Figure 6 shows the measured time per key of bitonic sort with 64 processors and 1M keys per processor for input sets with different entropies. The figure shows that the times for the communication steps and for the swap step are constant with respect to the distribution of the input set. As discussed in Section 3.3, the time for the local sort increases with entropy because of an increase in the TLB miss rate; however, this increase in time is offset by a decrease in the time for the merge sort step.[8] Therefore, the

[8]With low entropies, our implementation for finding the minimum element in the bitonic sequence is slower because more keys

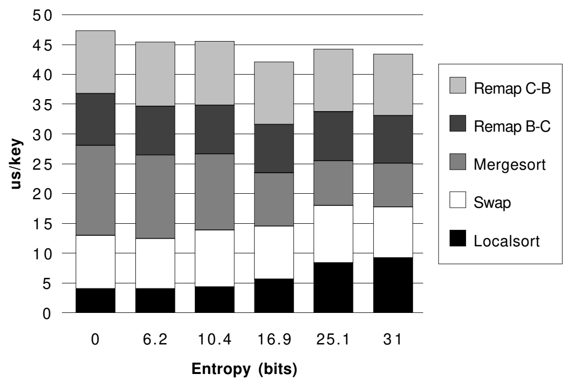

Figure 6: *Measured execution times per key on 64 processors for the phases of bitonic sort for different input key distributions.*

overall time for bitonic sort is relatively stable with different input key entropies, varying by only 12%.

5 Column Sort

Column sort [5], like bitonic sort, alternates between local sort and key distribution phases, but only four phases of each are required. Two key distribution phases use an all-to-all communication pattern and two use a one-to-one pattern. In column sort the layout of keys across processors is simple: the N keys are considered elements of an $n \times P$ matrix with column i on processor i. A number of restrictions are placed on the relative values of n and P, which require $N \geq P^3$.

The communication phases are *transpose*, *untranspose*, *shift*, and *unshift*, respectively. A local sort is performed on every column before each communication phase. Transpose is exactly the blocked-to-cyclic remap described in bitonic sort; untranspose is the cyclic-to-blocked remap. The shift permutation requires that the second half of the keys on processor i be sent to processor $(i + 1) \bmod P$. Thus, this step requires one-to-one communication. A small amount of local computation is performed to move the first half of the keys in each column to the second half of the same column. Similarly, in the unshift operation, the first half of the keys on processor i are sent to the second half of processor $(i - 1) \bmod P$. The local computation consists of moving the second half of each column to the first half.

When optimizing the local sort, it is essential to notice that after the first sort, the keys are partially sorted. In the second and third local sorts, there are P sorted lists of length n/P on each processor; therefore, a P-way merge could be performed instead of a general sort. However, empirical study showed that our general local radix sort is faster than a P-way merge sort for the P of interest. In the fourth sorting step there are two sorted lists of length $n/2$ in each column and a two-way merge is most efficient. The models for this local computation are once again found in Table 1.

must be examined before the direction of the sequence is determined, due to duplicate keys.

5.1 LogP Complexity

The execution time of column sort is the sum of its eight steps, alternating between local computation and communication.

$$\begin{aligned} T_{\text{columnsort}} &= n * t_{\text{localsort}} + (n * t_{\text{gather}} + T_{\text{remap}}) + n * t_{\text{localsort}} + T_{\text{remap}} \\ &\quad + n * t_{\text{localsort}} + T_{\text{shift}} + n * t_{\text{merge}} + T_{\text{unshift}} \\ T_{\text{shift}} = T_{\text{unshift}} &= \frac{n}{2} * t_{\text{shiftcopy}} + \left\lceil \frac{n}{2w} \right\rceil * \max(g, 2o) \end{aligned}$$

The time of the local sort is the same as in bitonic sort. Our implementation of transpose gathers the keys into contiguous memory before performing the remap. Untranspose does not require a scatter because the next step of the algorithm is a local sort on the column; therefore, untranspose is modeled as a single remap. The shift and unshift consist of a local copy of half of a column, receiving $n/2$ words from one processor, and sending the same amount to another. The communication has the same cost as a pairwise exchange. Since n is large compared to L, we simplify the model as shown above.

5.2 Empirical Results

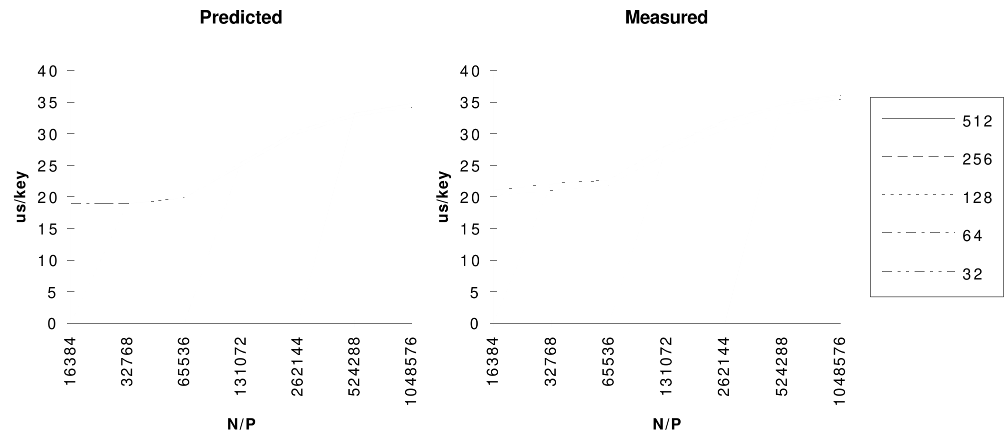

Figure 7: *Estimated and measured execution time of column sort on the CM-5. Note that, due to the restriction that $N \geq P^3$, our algorithm cannot sort all values of n for a given P.*

The predicted and measured time per key for column sort on uniformly distributed keys is shown in Figure 7. Note that not all data points are available for all numbers of processors, due to the restriction that $N \geq P^3$. The error between the predicted overall time per key and the measured time is less than 11% for all data points. The predicted time per key increases very little with an increasing number of processors; the measured time increases somewhat more, due to a rise in the gather time. The time per key increases more dramatically as the number of keys per processor grows. These two trends are in contrast with the behavior of bitonic sort, where time per key increased significantly with P, but slowly with n.

Figure 8 shows the predicted and measured times per key for each of the phases on 64 processors. This configuration was chosen because it is the largest on which the full range of keys can be sorted. These

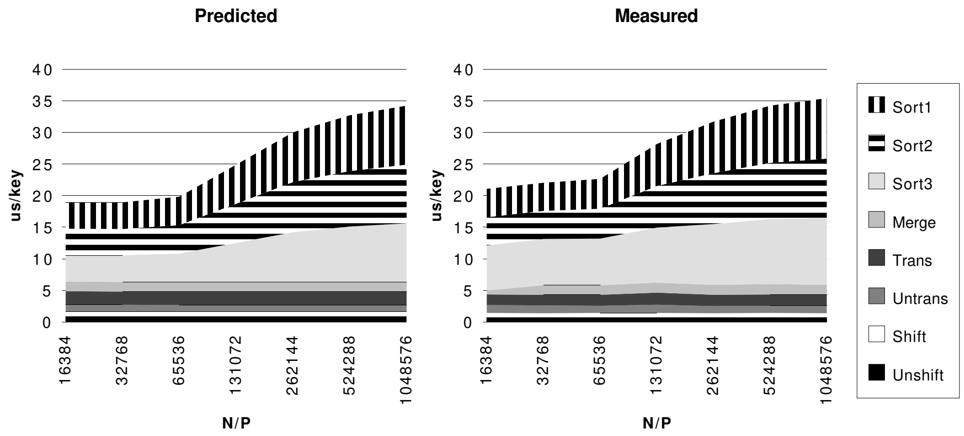

Figure 8: *Estimated and measured execution time of the phases of column sort on 64 processors.*

figures demonstrate that the time for the local computation steps increases with n. In fact, the increase in column sort is more dramatic than in bitonic sort. The time spent communicating remains constant with n, so the percentage of time spent communicating decreases with n, from 20% to 12%. The error between the predicted and measured values is negligible for the communication phases; the primary source of error is in our prediction for the local computation.

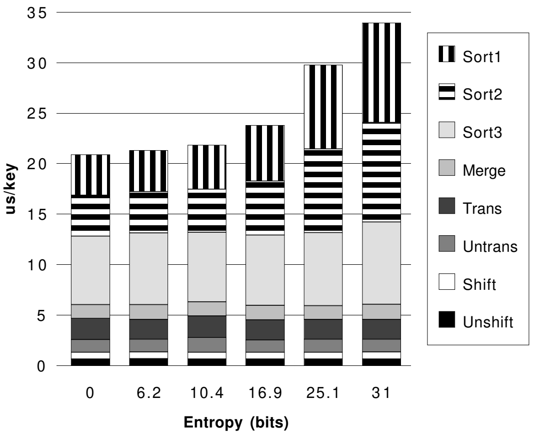

Figure 9: *Measured execution times per key on 64 processors for the phases of column sort for different input key distributions.*

The measured execution time for the phases of column sort for various input set entropies is shown in Figure 9. As expected, the time per key for the communication phases is constant; the time for the merge computation is also constant with entropy. However, because column sort performs three local sorts which each execute much faster at lower entropies, column sort is more than 60% faster for low entropy keys than for uniformly distributed keys and than our model predicts.

6 Radix Sort

Parallel radix sort requires fewer local computation and key distribution phases than the previous sorts; however, the communication phase is *irregular* and uses an additional setup phase to determine the destination of the keys. In the setup phase a global histogram is constructed, which involves a multi-scan and a multi-broadcast.

The parallel version of radix sort is a straight-forward extension to a local radix sort. In the parallel version, the keys begin in a blocked layout across processors. Each pass of the parallel radix sort consists of three phases. First, each processor determines the local rank of its digits by computing a local histogram with 2^r buckets. Second, the global rank of each key is calculated by constructing a global histogram from the local histograms. In the third phase, each processor has the global rank of the first of each digit, which is used to distribute each key to its proper position.

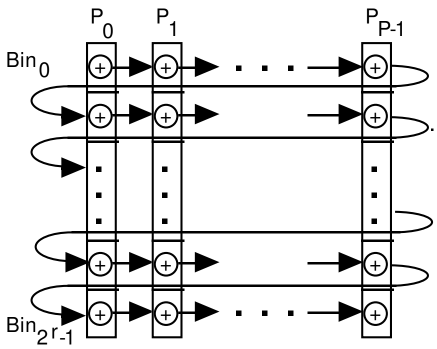

Figure 10: *Linear scan of local histograms to form the global histogram.*

The global histogram in the second step is constructed such that bucket i on processor p contains the sum of all local histogram buckets less than i on all processors and the local histogram buckets equal to i on processors less than p. In the naive approach to constructing the global histogram, each row of the global histogram depends upon the previous row. This construction is shown in Figure 10. When this dependency exists, a parallel prefix can be used for each row of bins, but the 2^r scans must be done sequentially. For a large radix, this is inefficient.

To eliminate the dependency, a *partial global histogram* is constructed by performing a scan on each bucket across processors. Thus, each row in the partial histogram is independent of the earlier rows. After the partial global histogram is constructed, the sum of each row of buckets, stored in the buckets on the last processor, is broadcast to all processors. Each processor computes its portion of the global histogram by adding the sum of the broadcasted buckets less than i to the bucket i in the partial histogram.

The algorithm just described consists of three components: a multi-scan to construct the partial global histogram, a multi-broadcast of the sum of each row, and finally, some local computation to adjust the values in the global histogram. The multi-scan and multi-broadcast communication operations are discussed further below.

In the key distribution phase, the processor and offset to which a key is sent depends upon the value in the global histogram; thus, this phase requires all-to-all *irregular* communication. Since the destination of the keys is dependent upon the value of the key, it is not possible to precompute a contention-free communication schedule as for the remap. Determining the expected slowdown due to contention of a random permutation under LogP is an interesting open problem. Our simulations and recent theoretical results[11] suggest that the slow-down is bounded by a small constant, but a thorough treatment of this problem is beyond the scope of this paper. Because of difficulties in modeling the contention in this phase, we elected to ignore it and model the time for this phase as

$$T_{\text{dist}} = n * \max(g, 2o + t_{\text{addr}}),$$

where t_{addr} is the time required to perform the address calculation before storing each element. [9]

6.1 Optimizing Multi-Scan and Multi-Broadcast

There are many different ways to tackle the multi-scan and multi-broadcast problems, but LogP provides valuable guidance in formulating an efficient pipelined approach. One might expect that a tree-based approach would be most attractive. However, for one processor to send n words to each of two other processors takes time at least $2n * g$. Thus, if n is much larger than P, it is faster to broadcast the data as a linear pipeline, than as a tree.[10] In a pipelined multi-scan, processor 0 stores its first value on processor 1; processor 1 waits until it receives this value, adds it to a local value, and passes the result to processor 2, and so on. Meanwhile, after sending the first value, processor 0 sends the second value to processor 1, and so on through each of the values. We concentrate on the multi-scan for the remainder of the discussion since multi-broadcast is identical to multi-scan, except the value is forwarded directly to the next processor without adding it to a local value.

A straight-forward estimate of the time for multi-scan is the time for one processor to forward $(2^r - 1)$ values to the next processor plus the time to propagate the last element across all processors. It takes time $\max(g, 2o + t_{\text{add}})$ to forward one element in multi-scan, so the time to perform a multi-scan over 2^r values should be

$$T_{\text{scan}} = (2^r - 1) * \max(g, 2o + t_{\text{add}}) + (P - 1) * (L + 2o + t_{\text{add}}).$$

In this analysis we are tacitly assuming that each processor receives a value, forwards it, and then receives the next value, so that a smooth pipeline is formed. This observation brings to light an important facet of the LogP model: time that a processor spends receiving is time that is not available for sending. In practice, receiving is usually given priority over sending in order to ensure that the network is deadlock-free. The difficulty in the multi-scan is that processor 0 only transmits, so it may send values faster than the rest of the pipeline can forward them. As a result, processor 1 receives multiple values before forwarding previous ones and processors further in the pipeline stall, waiting for data.

In order to maintain the smooth pipeline, the sending rate of processor 0 must be slowed to the forwarding rate of the others. The model does not specify the policy for handling messages at the processor, so in

[9] Our implementation of the key distribution does not take advantage of the size of the communication word. Modifications could be made to the algorithm such that keys destined for the same processor are first gathered and then stored in bulk.

[10] In [12] an optimal broadcast strategy is developed where the root sends each data element only once, but alternates among recipients in order to retain the logarithmic depth of a tree broadcast.

theory each processor could refuse to receive the next value until it has forwarded the present one and the capacity constraint would eventually cause processor 0 to slow to the forwarding rate. A simpler solution, which we chose to implement, is to insert a delay into the sending loop on processor 0 so it only sends at the forwarding rate $\max(g, 2o + t_{\text{add}})$.

This seemingly minor issue proved to be quite important in practice. Our initial naive implementation allowed processor 0 to send at its maximum rate. This was slower than our prediction by roughly a factor of two, which caused us to study the issue more carefully. After inserting a delay, the measured execution times were within 3% of that predicted by the model.

6.2 LogP Complexity

The total running time of the radix sort algorithm is the sum of the running time of the three phases multiplied by the number of passes. The optimal radix size r depends upon the number of keys to be sorted and the relative cost of creating the global histogram to the cost of permuting the keys. A larger radix implies more histogram buckets and thus a higher cost for creating the global histogram; however, less passes are performed. Our implementation uses a radix size of 16 bits.

$$T_{\text{radix}} = \left\lceil \frac{b}{r} \right\rceil * (T_{\text{localhist}} + T_{\text{globalhist}} + T_{\text{dist}})$$

$T_{\text{localhist}}$ involves only local computation; forming the local histogram requires initializing each of the histogram buckets and incrementing the corresponding bucket for each of the keys.

$$T_{\text{localhist}} = 2^r * t_{\text{zero}} + n * t_{\text{h}}$$

The LogP complexity of constructing the global histogram, is the sum of three components. Our models of the execution time of T_{scan} and T_{bcast} are those presented in Section 6.1, plus the cost of initializing each of the histogram buckets. T_{final} is the time to adjust the local histogram by the broadcasted values.

$$\begin{aligned}
T_{\text{globalhist}} &= T_{\text{scan}} + T_{\text{bcast}} + T_{\text{final}} \\
T_{\text{scan}} &= 2^r * t_{\text{zero}} + 2^r * \max(g, 2o + t_{\text{add}}) + (P-1) * (L + 2o + t_{\text{add}}) \\
T_{\text{bcast}} &= 2^r * t_{\text{zero}} + 2^r * \max(g, 2o) + (P-1) * (L + 2o) \\
T_{\text{final}} &= 2^r * t_{\text{bsum}}
\end{aligned}$$

6.3 Empirical Results

Figure 11 shows the predicted and measured times per key for radix sort on uniformly distributed keys. For small values of n, our measurements are only 9% higher than the prediction; for large values of n, our measurements are 37% higher. We predict that the number of processors has a negligible impact on the execution time per key, but measurements show that the execution time increases slowly with the number of processors due to a slight increase in key distribution time. The execution time per key decreases as the number of keys increases; therefore, while radix sort is slower for small n than both column and bitonic

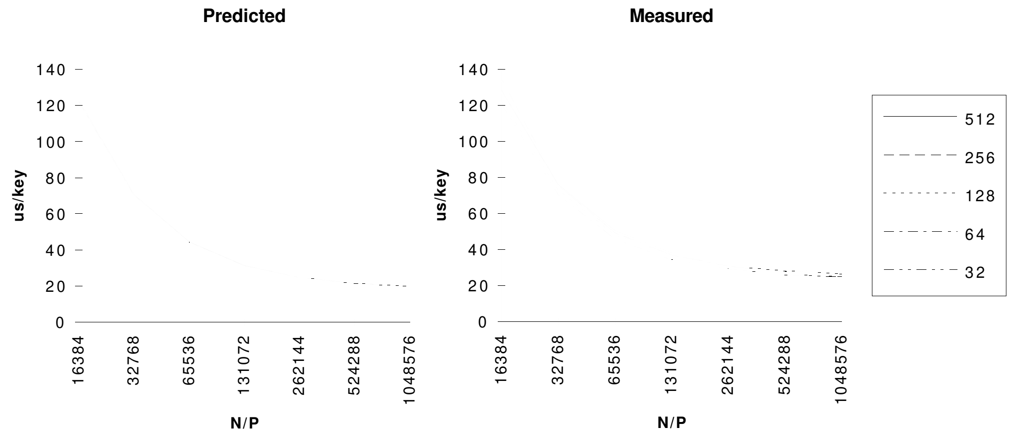

Figure 11: *Predicted and measured execution time per key of radix sort on the CM-5.*

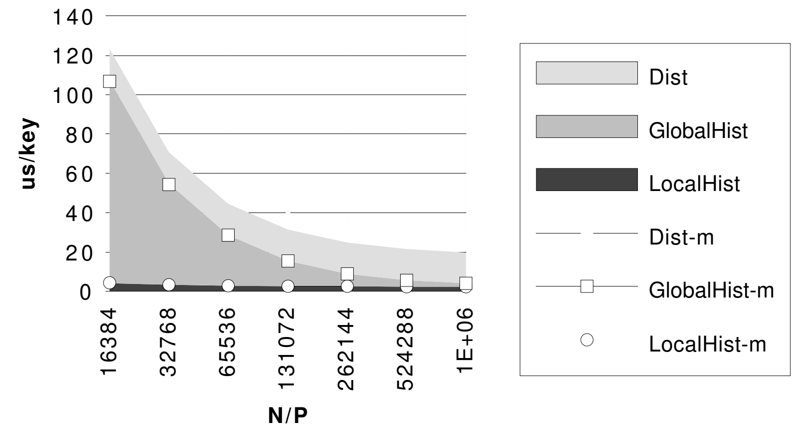

Figure 12: *Predicted and measured execution times per key of various phases in radix sort on 512 processors.*

sort, it is faster for large n.

Figure 12 shows the breakdown by phases for 512 processors. Both the prediction and the measurements show a significant decrease in time per key with increasing number of keys, due to the fixed cost of constructing the global histogram. This observation implies that a smaller radix size should be used to sort small data sets. At large values of n, the time to distribute the keys comprises 85% of the execution time. Our measurements show that the time to permute the keys across processors increases with both P and n. The model matches the measured execution time precisely for 32 processors and 16K keys. On 512 processors and 16K keys, our measured time is 34% higher than that predicted by the model; with 1M keys it is 47%. This increase in communication time both with the number of processors and problem size may be due to the contention which we do not model, although additional experiments indicate that the increase with n is largely due to tlb misses on the destination processor, similar to the misses observed in the local sort. As a result of the poorer accuracy of our model for very large numbers of processors, we may not be able to extrapolate our results to 1024 processors as well for radix sort as for the other sorts.

Figure 13 shows the execution time of radix sort for various input sets. The execution time for constructing the local histogram increases slightly with entropy, due to a decrease in the locality of histogram

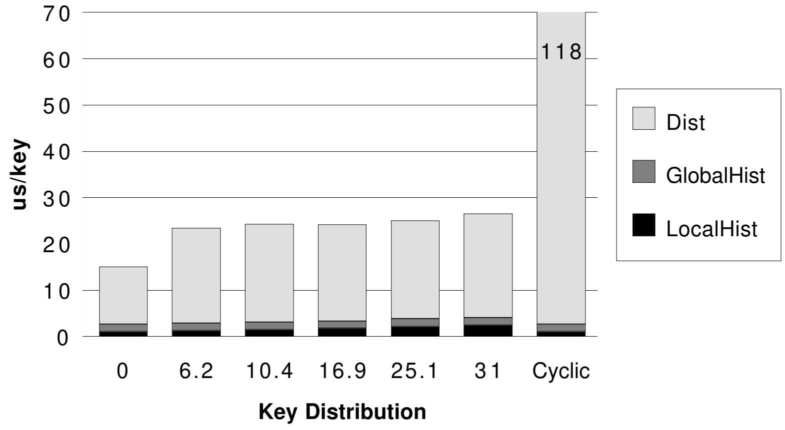

Figure 13: *Measured execution times per key on 64 processors for the phases of radix sort for different input key distributions. The time per key under the cyclically sorted input set extends beyond the graph.*

bucket accesses. The time for constructing the global histogram is constant with key entropy because the pipelined multi-scan and multi-broadcast are oblivious to the keys. In the key distribution phase, the communication pattern does depend upon the input set; however, the entropy of the keys is of lesser importance than the layout of keys across processors. The key distribution phase executes 5% faster with an input set of entropy 25.1 than one of 31; however, this is due not to changes in the communication schedule, but to increased locality in the histogram bucket accesses when calculating the destination addresses.

In contrast, an input set of a constant value (an entropy of 0) is already sorted on each of the passes; thus, during key distribution, each processor stores keys only to itself.[11] Not only does this guarantee that no contention occurs, but processors compute for some time less than o to perform the local store; this results in a communication phase which is 80% faster than with uniformly distributed keys.

In [9], similar results are presented for the execution time of the phases in radix sort as the entropy of keys is varied: a decrease in the total execution time as the entropy decreases, with a marked decrease with an entropy of 0. While the shape of their curve and ours are very similar, their execution times are between 50% and 66% faster. Their improvement in execution time occurs in the key distribution phase; if our implementation were changed to pack multiple keys into a single communication word, this difference might be reduced.

Figure 13 also shows the performance for the worst case input set: the keys are initially sorted on each pass cyclically across processors. On every pass, each processor first stores n/P keys on processor 0, then each stores n/P keys on processor 1, and so forth. This contention in the communication schedule slows down the key distribution phase more than five times that for uniformly distributed keys.

[11] Note that not all input sets which are initially sorted cause processors to store only to themselves, since the keys must be in sorted order for each of the $\lceil \frac{b}{r} \rceil$ passes.

7 Sample Sort

An interesting recent algorithm, called sample (or splitter) sort [3, 13], pushes the pattern of alternating phases of local computation, destination setup, and key distribution to the extreme — it performs only one of each. The key distribution phase in sample sort exhibits the most complicated structure of any in the four sorts: *irregular*, *unbalanced* all-to-all communication.

The idea behind sample sort is as follows. Suppose we could pick every n-th key in the final sorted order; these $P - 1$ values split the data into P equal pieces. If each processor has the $P - 1$ splitters, it can send each key to the destination processor. After sending and receiving all of its keys, each processor sorts its keys locally. In sample sort, rather than picking precisely every n-th key, the splitters are guessed by sampling the initial data. Some $s * P$ elements are selected at random, sorted, and every s-th element is selected.

This leads to the three phases of sample sort. In the setup phase, the *splitter* step, every processor sends s of its keys to processor 0, where s is the *sample size*. Processor 0 sorts the samples, selects keys $s, 2s, ..., (P-1) * s$ as splitters, and broadcasts the splitters to the other processors. In the second step, the *distribute* phase, every processor sends each of its keys to the correct destination processor, as determined by a binary search on the splitter array.[12] In the last phase, a local radix sort is performed on the received keys.

The key distribution step of sample sort involves irregular, *unbalanced* all-to-all communication, *i.e.*, each processor potentially receives a different number of keys. We call the ratio of the maximum number of keys received by a processor to the average the *expansion factor*, E. Assuming a large sample size, a large number of keys per processor, and a random input key distribution, the expansion factor can be bounded by a small constant with high probability [3]. In our analysis of the distribution of keys in radix sort, we ignored the destination contention that occurs when multiple processors send to the same destination processor. The distribution phase for sample sort is similar, so we continue to ignore the potential contention. However, because the communication is unbalanced, one processor may receive up to $E * n$ keys. The execution time of this step is limited by the processor receiving the most keys.[13]

7.1 LogP Complexity

Under LogP, we model the time to perform the sample sort as the sum of the three phases.

$$T_{\text{sample}} = T_{\text{split}} + T_{\text{dist}} + T_{\text{localsort}}$$

The time for the first phase, the splitter step, is the sum of its three components: collecting the samples and sending them to processor 0, sorting the samples on processor 0, and broadcasting the splitters. In our

[12] With low entropy input sets, there is a high probability that the splitters are not unique; special care is taken to distribute the keys with those values evenly across the processors.

[13] As with the key distribution in radix sort, our implementation does not use a bulk communication style. Once again, additional computation could be performed to gather keys to take advantage of the communication word. For example, instead of independently determining the destination processor of each key and then storing each key, the keys could be first sorted locally on each processor, effectively gathering the keys destined for the same processor, and then storing the keys in bulk.

implementation, we use a sample size of $s = 64$. Processor 0 sorts the samples using an eight-bit radix sort and broadcasts the splitters, using the multi-broadcast described in Section 6.1 for P values.

$$\begin{aligned}
T_{\text{split}} &= T_{\text{collect}} + T_{\text{splitsort}} + T_{\text{bcast}} \\
T_{\text{collect}} &= \left\lceil \frac{s}{w} \right\rceil P * g + L \\
T_{\text{splitsort}} &= s * P * t_{\text{localsort_8}} \\
T_{\text{bcast}} &= P * t_{\text{zero}} + P * \max(g, 2o) + (P-1) * (L + 2o)
\end{aligned}$$

In the distribution phase the execution time is limited by the processor receiving $n * E$ keys. Before sending each key, each processor must perform a binary search on the splitter array to determine the destination processor. We model the time of this search for each key, $t_{\text{search}} = \lg P * t_{\text{compare}}$, as the number of comparisons multiplied by the time to perform a single comparison. Because this lookup time is larger than g for a large number of processors, the communication is spread further apart and the destination contention is less of an issue than in radix sort.

$$T_{\text{dist}} = n * \max(E * g, o + t_{\text{search}} + E * o)$$

The local sort time in sample sort is different than that in bitonic and column sort because the number of keys being sorted on a processor may be as large as $E * n$. This affects not only the total sorting time, $T_{\text{localsort}} = E * n * t_{\text{localsort}}$, but also $t_{\text{localsort}}$, the sorting rate per key.

For our sample size, number of keys per processor, number of processors, and uniformly distributed input keys, we found that $1.22 < E < 1.45$. In our model, we approximate E with the mean of the observed expansion factors: 1.33. Note that after sample sort has finished sorting the keys, the number of keys per processor is not constant. If this condition is desired, then an extra phase is necessary to redistribute the keys evenly across processors.

7.2 Empirical Results

The predicted and measured times per key for sample sort on uniformly distributed keys are displayed in Figure 14. On 512 processors, our measurements are accurate within 4% of that predicted by the LogP model. From the plots, we see that the time per key increases with the number of processors, for reasons discussed below. The behavior with increasing n depends upon the number of processors: for small P, it increases slightly; for large P, it decreases more significantly. The measured execution times of sample sort exhibit variation with n because the actual expansion factors are dependent on the random data sets.

The increase in time per key with the number of processors occurs in two different phases. The time of the splitter phase increases because with more processors, there are more samples to sort on processor 0 and more splitters to distribute across processors; however, the effect of this startup cost diminishes as n increases. The time of the key distribution phase increases because the search on the splitters takes time proportional to $\lg P$; this cost does not diminish with n.

The predicted and measured execution time per key on 512 processors is plotted in Figure 15. The

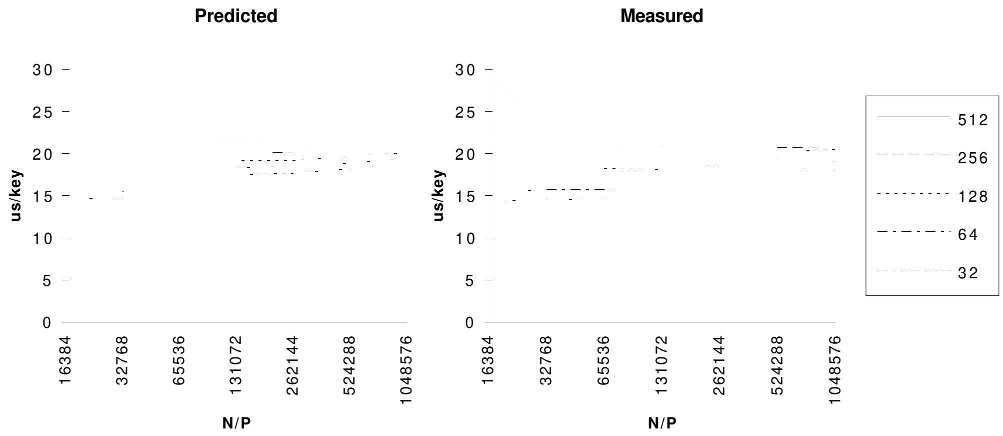

Figure 14: *Estimated and measured execution time of sample sort on the CM-5.*

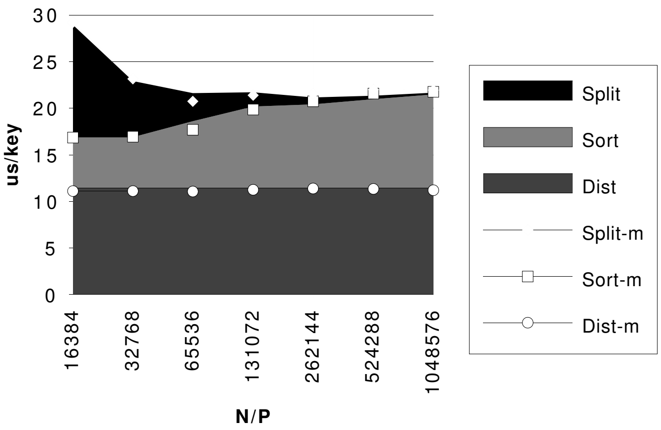

Figure 15: *Estimated and measured execution times of various phase of sample sort on 512 processors.*

measured time for the key distribution phase matches the predicted time very closely and is constant with n because the observed expansion factor on 512 processors varies very little from 1.33. However, the percentage of time spent distributing keys increases from 38% to 51% as the number of keys increases. Once again, the time for the local sort is greater for larger data sets due to cache misses. The execution time of the splitter phases decreases dramatically with the number of keys per processor. From this figure, it is obvious that the cost of the splitter phase is too large for small values of n: on 512 processors, with an oversampling ratio of 64, the splitter phase sorts 32K samples. The time for this phase could be reduced by sorting the samples on multiple processors, by using an 11-bit radix sort rather than the eight-bit radix sort, or by using a smaller oversampling ratio.[14]

Figure 16 shows the execution time per key of the phases of sample sort for various input key distributions. The time for the splitter phase is negligible for 64 processors and 1M keys per processor for all keys distributions. Once again, the time for the local sort increases with the entropy of the keys. As was the case with radix sort, if constant-valued keys are ignored, then the execution time of the key distribution phase

[14]Of course, a smaller value of s results in higher expansion factors.

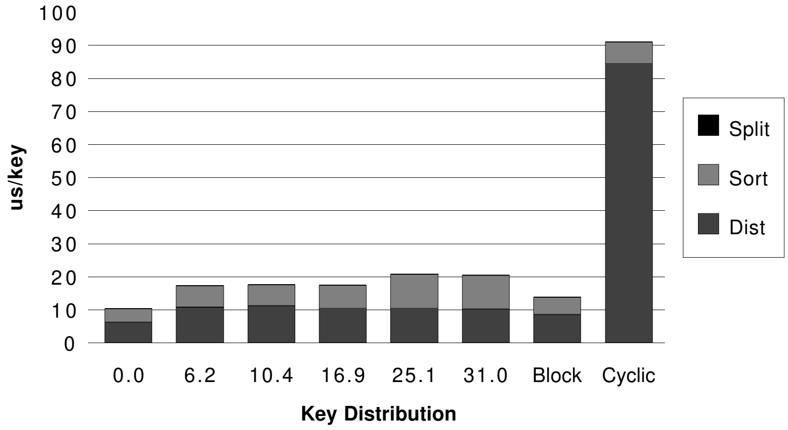

Figure 16: *Measured execution times per key on 64 processors for the phases of sample sort for different input key distributions.*

exhibits very little variation with key entropy. The time per key is slightly greater at the lower entropies because the expansion factors are larger.

The layout of keys across processors is more significant to the execution time of the key distribution phase. For example, distributing constant-valued keys is 63% faster than distributing keys with an entropy of 31 for two reasons: first, our implementation of the binary search is faster when a key matches one of the splitter values; second, each of the keys is stored on the sending processor. The figure shows that when the key values are unique, but already sorted across processors, the distribution phase is only 20% faster than for keys with an entropy of 31; this smaller speedup occurs because the binary search time remains unchanged from the uniform-distribution case, but keys are not moved across processors.

Finally, the worst case key distribution time occurs when the keys are initially sorted cyclically across processors. The contention in the communication results in a phase that is more than eight times slower than that for uniform keys. This slowdown due to contention is similar to that observed in radix sort for its worst case layout; however, the layout of keys that produces the most contention in radix sort is not a simple cyclically-sorted list and seems much less likely to occur in real input sets.

8 Comparison

The accuracy of the model encourages us to extrapolate to configurations which we have not yet measured. Figure 17 shows the predicted execution time of the four sorts for uniform data sets on 32 and 1024 processors. Note that with 1024 processors, column sort, because of its layout restrictions, is unable to sort any of the data sets of interest and so is not included in the figure. It appears that two of the sorts, radix and sample sort, are fast enough on 1M keys per processor and 1024 processors to sort one billion keys in less than half a minute, since they achieve a rate of about $25 \mu s$ per key.

The predicted time per key is slightly less for radix sort than for sample sort; however, as mentioned in Section 6.3, for a large number of processors and large problem sizes the predictions for radix sort are less accurate, which may lead to a 40% higher execution time than predicted. Both of these sorts have the

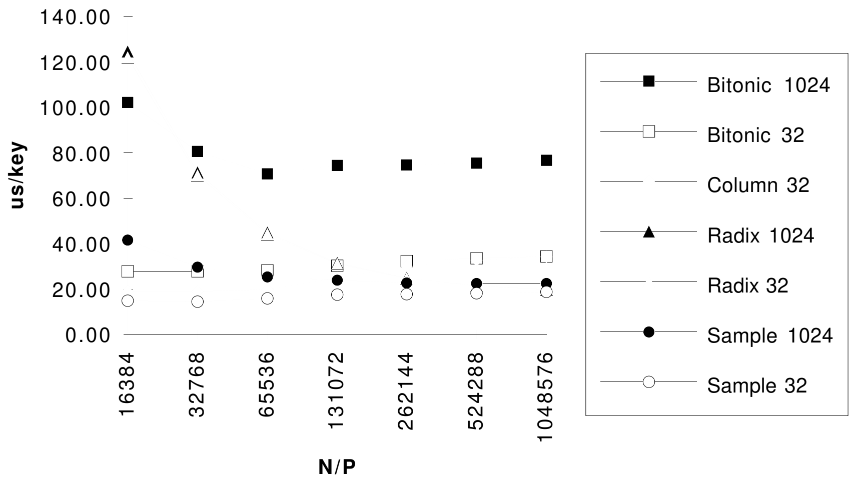

Figure 17: *Estimated execution time of four parallel sorting algorithms under LogP with the performance characteristics of the CM-5.*

disadvantage that the time for the key distribution phase is extremely sensitive to the layout of the keys across processors; each has a worst case layout that leads to an execution time that is five to eight times slower than that for uniform data. The worst case layout for sample sort (a cyclically sorted list) may occur more frequently in real input sets.

A question of interest is the execution time of the algorithms for the four combinations of small and large values of n and of P. As discussed above, for a large number of processors and large problem sizes, the performance of radix and sample sort are very similar; the choice between the two may depend on the input key distribution. On a small number of processors and small data sets, column and sample sort have nearly identical performance. Again, the decision for choosing between these two sorts would be based upon the expected distribution of the input keys and whether the number of keys meet the constraints imposed by column sort. With a small number of processors and large data sets, radix and sample sort once again have similar performance, with the tradeoff depending on the input key distribution. Finally, with small data sets on a large number of processors, sample sort significantly outperforms the other sorts.

9 Summary

In this paper, we have analyzed the performance of four parallel sorting algorithms.[15] We have found that when sorting algorithms are highly optimized to exploit the layout of keys across processors that they consist of alternating phases of local computation, setup to determine the destination of keys, and key distribution.

LogP captures the characteristics of modern parallel machines with a small set of parameters: latency (L), overhead (o), bandwidth (g), and number of processors (P). We have shown that the LogP model is a good predictor of the communication performance. In order to accurately predict the total execution of the algorithms, we required a model for the local computation; because LogP does not specify how local

[15]The Split-C implementations of the four sorting algorithms, as well as the Split-C compiler, are available by anonymous ftp from ftp.CS.Berkeley.EDU.

computation is modeled and because this was not the focus of our study, our model of the local computation is based on measurements. We discovered that a more elaborate model was needed for the local computation phases than for the communication phases in order to predict the two with comparable accuracy.

One issue when predicting execution time is whether the expected execution time or the worst case execution time should be used. The expected execution time depends upon both the distribution and the layout of the input keys. An open question remains as to how to characterize the input keys. We found that the execution time of the local radix sort was very sensitive to the probability distribution, or the entropy, of the input keys; our model of the local sort accurately predicts only the execution time for uniformly distributed keys. The key distribution phases in radix and sample sort exhibited substantially different performance with different layouts of keys across processors, due to differences in contention. Contention is modeled in LogP by the capacity constraint; however, because predicting the expected number of keys destined for a particular processor is difficult, we chose to ignore the contention in these phases in our model. Therefore, our model focused on predicting the expected execution time for random, uniformly distributed keys.

We have also shown that LogP is a valuable guide in illuminating deficiencies in implementations. Specifically, when the measured execution time of the all-to-all remap and the pipelined multi-scan did not match the execution time predicted by the model, we examined our implementations more carefully and found that we were violating constraints specified by the model. Re-implementations corrected the violations and brought the measured execution times to within 15% and 3%, respectively, of the predictions.

Acknowledgement

The authors wish to acknowledge the computational support provided by the NSF Infrastructure Grant number CDA-8722788 and the Advanced Computing Laboratory of Los Alamos National Laboratory. David Culler is supported by an NSF Presidential Faculty Fellowship CCR-9253705 and LLNL Grant UCB-ERL-92/172. Andrea Dusseau is supported by an NSF Graduate Research Fellowship and Siemens. Klaus Erik Schauser was supported by an IBM Graduate Fellowship. Richard Martin is supported by a UC Berkeley Fellowship. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] 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.

[2] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel Programming in Split-C," in *Supercomputing*, 1993.

[3] G. Blelloch, C. Leiserson, and B. Maggs, "A Comparison of Sorting Algorithms for the Connection Machine CM-2," in *Symposium on Parallel Algorithms and Architectures*, July 1991.

[4] K. Batcher, "Sorting Networks and their Applications," in *Proceedings of the AFIPS Spring Joint Computing Conference*, 1986.

[5] T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting," *IEEE Transactions on Computers*, Apr. 1985.

[6] M. Zagha and G. Blelloch, "Radix Sort for Vector Multiprocessors," in *Supercomputing*, 1991.

[7] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, "The Stanford Dash Multiprocessor," *IEEE Computer*, vol. 25, pp. 63–79, Mar. 1992.

[8] C. Shannon and W. Weaver, *The Mathematical Theory of Communication*. University of Illinois Press: Urbana, 1949.

[9] K. Thearling and S. Smith, "An Improved Supercomputer Sorting Benchmark," tech. rep., Thinking Machines Corporation, 1991.

[10] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," in *Proc. of the 19th International Symposium on Computer Architecture*, May 1992.

[11] P. Liu, W. Aiello, and S. Bhatt, "An atomic model for message-passing," in *Symposium on Parallel Algorithms and Architectures*, 1993.

[12] R. Karp, A. Sahay, E. Santos, and K. E. Schauser, "Optimal Broadcast and Summation in the LogP Model," in *5th Symp. on Parallel Algorithms and Architectures*, June 1993.

[13] J. H. Reif and L. G. Valiant, "A Logarithmic time Sort for Linear Size Networks," *Journal of the ACM*, vol. 34, pp. 60–76, Jan. 1987.