1

Fast Parallel Sorting under LogP: from theory to practice

DAVID E. CULLER, ANDREA C. DUSSEAU, RICHARD P. MARTIN, KLAUS ERIK SCHAUSER

1.1 ABSTRACT

The LogP model 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). In this paper, we analyze four parallel sorting algorithms (bitonic, column, radix, and sample sort) under LogP. We develop implementations of these algorithms in a parallel extension to C and compare the actual performance on a CM-5 of 32 to 512 processors with that predicted by LogP using parameter values for this machine. Our experience was that the model served as a valuable guide throughout the development of the fast parallel sorts and revealed subtle defects in the implementations. The final observed performance matches closely with the prediction across a broad range of problem and machine sizes.

1.2 INTRODUCTION

Fast 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. In current (1993) technology, "fast parallel sorting" corresponds to a practical performance target of "sorting a billion large keys on a thousand processors

Book title and editor name

in less than a minute." This problem size is large enough that algorithmic complexity matters and large scale parallelism is required. Our target is a practical because a modern RISC workstation can sort one million 32-bit keys in about 10 seconds.

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 [Cul93a], which attempts to capture the key performance characteristics of modern large scale multiprocessors. In particular, the model reflects the technological reality that these machines are essentially a collection of workstation-class nodes connected by a dedicated, high performance network. The LogP model downplays the role of the topology of the interconnection network and instead describes only its performance characteristics. The individual processors have substantial computing power and memory capacity, and the total number of processors is limited. Thus, the most interesting problems have many data elements per processor. Certainly one of the important facets of a parallel sorting algorithm is how it exploits the grouping of data within processors. We find that fast sorting algorithms under LogP alternate between phases of local computation and global communication, which may involve a general transformation of the entire data set.

In the sections that follow, we examine four sorting algorithms: bitonic sort [Bat68], column sort [Lei85], radix sort, and sample sort [Ble91]. 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. We develop an efficient implementation of the algorithm in a parallel language, Split-C, on the Thinking Machines CM-5 and compare the predictions of the model with the measured performance.

Split-C provides an attractive basis for the study, because it exposes the capabilities allowed under 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 straight-forward 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 [Ble91]. That work developed an informal model of the CM-2 with a large number of small processors, a specific hypercube interconnection structure, and hardware support for scan operations. We augment the comparison to include column sort and address a more general setting, formalized by LogP. Finally, the algorithms are implemented in a language that can be supported on a variety of machines.

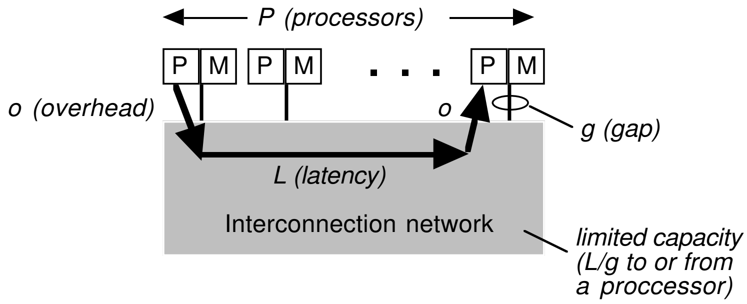

Figure 1.1 *The LogP model describes an abstract machine configuration in terms of four performance parameters: L, the latency experienced in each communication event, o, the overhead experienced by the sending and receiving processors for each communication event, g, the gap between successive sends or successive receives by a processor, and P, the number of processors/memory modules.*

1.3 LogP

The LogP model 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. This convergence is driven by a variety technological and economic factors, including the tremendous growth in cost-performance of workstations and personal computers. These factors also limit the number of processors of practical interest to a few thousand, making $P \ll N$ the regime of interest.

1.3.1 Model parameters

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

L: an upper bound on the *latency*, or delay, incurred in communicating a message containing a small 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 specificied by the model

The model assumes that all messages are of a “small size”, which we will call the communication word and denote w. 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 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 (e.g., by not polling the network or not responding to interrupts), 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. Although the model does not explicitly specify the characteristics of the local processor, it assumes that optimizing local processor performance is important. Where communication does occur, its latency can be masked by simultaneous use of the processor, within the bandwidth limits of the network.

1.3.2 Use of the model

To demonstrate the use of the model, let us consider some simple interprocessor operations. As illustrated in Figure 1.1, 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 global 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 we can ignore g. 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 with an interval of time g. These each take time L to reach the destination, and the destination processor is busy for time o after receiving each. If n is large, then the factor of $2o$ and L can be safely ignored. Observe that the processors spend time $n * o$ performing the actual communication, with the remaining time available for other computation. Typically this is utilized for address calculations, memory transfers, or calculations associated with each message.

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

[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[Len92].

have been sent, each processor receives messages at the sending rate. 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. Once again, if n is large this equation can be approximated with $n*\max(2o,g)$.

In many cases, it is useful to describe the communication time in terms of computational words transfered, recognizing that w words can be transferred in a single message. For example, the time for pairs of processors to exchange n computational words is

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

However, when the computational words are to be scatttered across destination processors it may not be possible to take advantage of the packing of computational words into a communication word. Another common communication pattern is for each processor to send n/P words to each other processor. We will look at several variations on this case for specific sorting algorithms.

1.3.3 Experimental Platform

Our measurements are performed on the CM-5, a massively parallel MIMD computer based on the Sparc processor. Each node consists of a 33 Mhz Sparc RISC processor chip-set (including FPU, MMU and 64 KByte direct-mapped write-through cache), 32 MBytes of local DRAM memory and a network interface. The nodes are interconnected in two identical disjoint incomplete fat trees, and a broadcast/scan/prefix control network. The implementation of the sorting algorithms do not use the vector accelerators.

In previous experiments on the CM-5 [VEi92, Cul93a] we determined that $o \approx 2.2\mu s$ and, on an unloaded network, $L \approx 6\mu s$. The communication word size, w, is equal to four (32-bit) processor words. The bisection bandwidth is 5MB/s per processor, so we take g to be $4\mu s$.[2]

1.3.4 Implementation Language

Our sorting algorithms are written in Split-C [Cul93b], 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. The layout of data across processors can be specified by *spread arrays*. Thus, programs can be optimized to take advantage of the grouping of data onto processors. 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.

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

The unusual aspects of Split-C are the assignment operations it provides to allow the programmer to overlap communication and avoid unneccessary communication events. With *split-phase* assignment, expressed with `:=`, the initiating processor does not wait for the response, so it can perform $L - 2o$ cycles of useful work during the remote operation. In particular, it can initiate additional requests. A *signalling store*, expressed with `:-`, is not acknowledged to the initiating processor, so it is a one-way operation and takes time $2o + L$. Bulk transfer of multiple words is provided for all of the described styles of communication.

1.4 PARALLEL SORTS

In the following sections we examine four sorting algorithms: bitonic, column, radix and sample sort. We predict the execution time of the algorithms based on the parameters of the LogP model and a small set of parameters that characterize the local computational performance of Thinking Machines CM-5. We implement the algorithm in Split-C and compare the measured execution times of the sorts to the model predictions on configurations ranging from 32 to 512 processors. Special attention is given to the more challenging communication phases. In keeping with our goal of sorting one billion 32-bit keys in less than a minute (*i.e.*, 60 μs per key per processor with 1M keys on each of 1024 processors), we are interested in the performance of the algorithms when the number of keys per processor is large, specifically between 16K and 1M. Throughout, we use P as the number of processors, numbered 0 through $P-1$, N as the total number of keys to be sorted, and $n = \frac{N}{P}$ as the number of keys per processor.

Each of the parallel sorts involves phases of purely local computation, which exploit the layout of keys, phases of global communication, which involve redistributing the keys across processors, and intermediate phases of setup to determine the destinations of the keys. Interestingly, the sorts with the fewest communication phases are also the sorts with the most complex communication pattern and intermediate setup. The order in which we discuss the sorts is related to the complexity of their communication phases.

Bitonic sort uses only *regular*, *balanced* all-to-all communication, *i.e.*, the communication schedule is oblivious to the values of the keys and each processor receives the same amount of data as it sends. A contention-free schedule can be constructed *a priori*. However, there are $2 \lg P$ phases. Column sort contains only four regular, balanced communication phases: two involve pairs of processors, the other two require all-to-all communication. However, the algorithm requires that $n = \frac{N}{P}$ increase rapidly with P. Radix sort performs only two communication phases (for 32-bit keys and a 16-bit radix), but a multi-scan and multi-broadcast across processors is needed to determine the communication pattern. The key distribution phase involves *irregular*, balanced all-to-all communication. Sample sort performs only one global communication phase, but a complex intermediate setup phase is used to determine the destinations of keys and the communication is both irregular and *unbalanced*. With *unbalanced* communication it is no longer guaranteed that each processor contains exactly n keys after the distribution phase.

1.5 BITONIC SORT

In this section, we discuss a variant of Batcher's bitonic sort [Bat68]. After describing the general algorithm, we present a data layout that reduces the communication across processors and enables optimizations for the local computation. We then describe how the LogP model guides us to efficient implementations 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.[3] The basic algorithm for sorting N numbers performs $\lg N$ *merge stages*, each consisting of multiple *bitonic merges*. A bitonic merge rearranges a bitonic sequence into a single sequence in monotonic order. During the i-th merge stage, where $0 < i \leq \lg N$, $N/2^i$ monotonic ordered sequences of length 2^i are formed from bitonic sequences of length 2^i. The monotonic sequences are ordered such that two neighboring sequences (one monotonically increasing and one monotonically decreasing) can be combined to form a new bitonic sequence for the next merge stage. Thus, the first merge stage produces $N/2$ monotonic ordered sequences of length two from the initial set of keys. After the final merge stage only a single monotonically increasing sequence of length N remains.

The communication structure of a bitonic merge of size 2^i can be represented by a butterfly 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, one per merge stage, as suggested by Figure 1.2. The i-th merge stage consists of i *merge steps*.

1.5.1 Optimizing Data Layout

The algorithm described so far 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 simply simulate the individual steps in the butterfly. However, we derive a more efficient data placement and work assignment that was inspired by the mapping used for large FFTs, as discussed in [Cul93a]. Consider two layouts: blocked and cyclic. Under a blocked layout, the first $n = N/P$ keys are assigned to the first processor, the second n keys to the second processor, and so on. The first processor is responsible for the operations represented by the first n rows of butterfly nodes, the second processor for the second n rows, and so on. Under this layout, the first $\max(0, i - \lg n)$ merge steps of the i-th merge stage require a remote reference for each node, while the last $\min(i, \lg n)$ steps are completely local. For example, with four processors the first two merged stages in Figure 1.2 are entirely local and the first step of the last stage requires remote accesses.

Under a cyclic layout the communication requirements are reversed. Here the first key (and first row of nodes) is assigned to the first processor, the second key (and second row of nodes) to the second processor, and so on. Thus, the first $\min(\lg n, \max(0, i - \lg n))$ steps of the i-th merge stage are completely local, while the

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

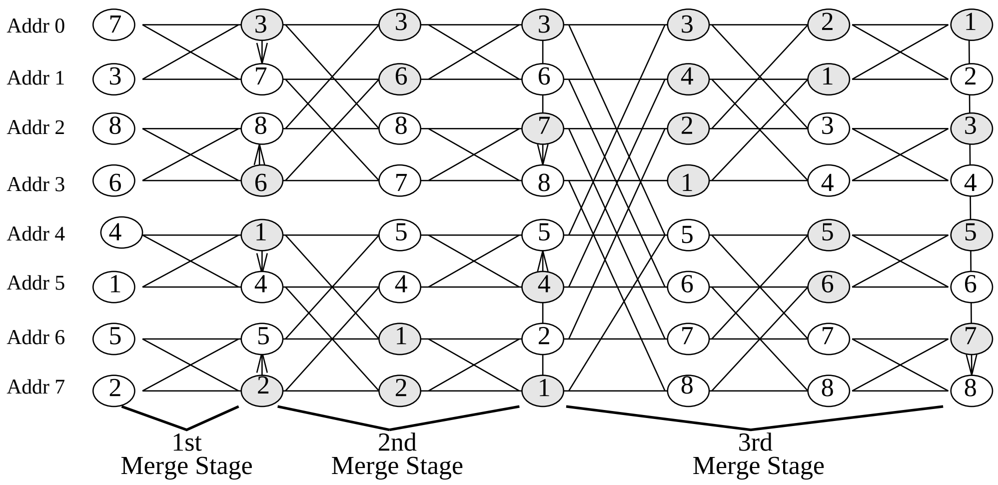

Figure 1.2 *The communication required for bitonic sort to sort eight keys using a series of merge stages. A row of nodes represents an address containing one of the keys. The edges show which keys are swapped with one another. A shaded node designates an address where the minimum of the two keys is placed. The arrows indicate the monotonic ordered sequences, with the arrowhead pointing towards the largest key.*

remaining steps all require a remote reference for each node. Note that this results in a higher communication complexity than a blocked layout because the number of steps is small for the early stages.

The communication requirement can be substantially reduced by periodically remapping the data from a blocked layout to a cyclic layout and vice versa. Our bitonic sort starts with a blocked layout. Thus, the first $\lg n$ merge stages are entirely local. For subsequent merge stages, we remap from a blocked to a cyclic layout, compute $i - \lg n$ merge steps locally, remap back into a blocked layout, and perform the last $\lg n$ merge steps locally. Assuming $n > P$, communication within each merge stage involves only the two remap operations.

Observe that with a fixed layout, either blocked or cyclic, each remote step in a stage involves exchanging n words between pairs of processors, whereas in the optimized layout the entire stage involves two all-to-all communications of n words per processor. As we will see later, all-to-all communication can be implemented nearly as efficiently under LogP as one-to-one communication. A gather and 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.

1.5.2 Optimizing Local Computation

Having optimized the data layout to minimize the communication, we can optimize the local computation to exploit the grouping of data elements. The simplest implementation of a local bitonic merge step mimics the computational structure of the butterfly *within* each processor, *i.e.*, for each node in the butterfly, it compares two keys and selects either the maximum or minimum. However, the purpose of the first $\lg n$ merge stages is simply to form a monotonically increasing or decreasing sequence

Variable	Operation	Time Per Key (μs/key)	
t_{swap}	simulate cyclic butterfly for key	$-0.08 + 0.025 * \lg n$	
$t_{\text{mergesort}}$	sort bitonic sequence of keys	1.0	
t_{scatter}	move key for cyclic to blocked remap	0.46	if $n \leq$ 64K
		$0.44 + 0.00059 * P$	otherwise
t_{gather}	move key for blocked to cyclic remap	0.52	if $n \leq$ 64K or $P \leq 64$
		1.1	otherwise
$t_{\text{localsort}}$	local radix sort of randomly distributed keys	4.1	if $n <$ 64K
		$-22.7 + 1.7 * \lg n$	if 64K $\leq n \leq$ 256K
		$-0.7 + 0.5 * \lg n$	if 256K $< n$

Table 1.1 Models of local computation in bitonic sort.

of n keys on each processor. Therefore, we can replace all of these merge stages with a single, highly optimized *local sort*.[4]

Additional optimizations can be applied to the final $\lg P$ merge stages. For the merge steps performed under a cyclic layout, the processor performs a comparison and conditional swap of pairs of keys, rather than comparing the keys twice to individually select the minimum and maximum. The purpose of the final $\lg n$ steps performed under a blocked layout is again to produce sorted sequences of the n keys on every processor. Moreover, since the keys on each processor form a bitonic sequence, we can use a simple *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. Finally, we combine the merge sort with the gather of the next remap phase by arranging the keys in the gathered order.

Since LogP does not specify how local computation is to be modeled, we chose to assign a time per key per processor to each local computation. This time was determined by measuring the execution time of each computation on the relevant range of keys on a single processor. When possible, we use a constant rate for the time per key. However, as can be seen in Figure 1.3, we found that the local sort exhibited three performance regimes, due to cache effects. Thus, we use a piecewise linear model for the local computation times, as described in Table 1.1. The models for the gather and scatter time are dependent upon P due to cache effect, since the stride with which keys are accessed varies with the number of processors.

1.5.3 Optimizing Regular Balanced Communication

The remaps between cyclic and blocked layouts involve *regular* and *balanced* all-to-all communication. The remap operation consists of a series of iterations, where on each iteration the processor performs a gather or scatter and exchanges keys with another

[4] This local sort also plays an important role in in column and sample sort, accounting for up to 80% of the execution time. Thus, it is important to optimize the local sort carefully. We determined empirically that *radix sort* is faster than quicksort and other local sorts for our range of number of keys per processor. We also determined that an 11-bit radix sort is faster than an eight-bit or 16-bit radix sort.

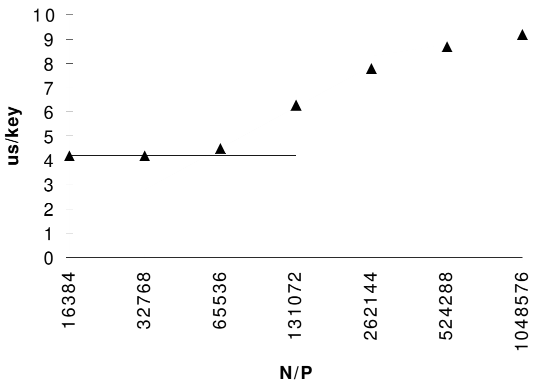

Figure 1.3 *Measured execution time per key and piecewise linear model of local sort. The black triangles indicate the measured rates for the 11-bit radix sort. The three lines represent the piecewise linear model used for the LogP analysis. They reflect the three operating regimes of the local sort: completely in-cache ($N/P < 64K$), completely out-of-cache ($N/P > 256K$), and in-between.*

processor. P-1 iterations are required for each processor to exchange keys with every other processor. Assuming there are no send stalls and excluding the time to perform gather or scatter operations, the execution time of the remap should be simply $(P-1)$ iterations of pair-wise exchanges of n/P keys.

$$
\begin{aligned}
T_{\text{remap}} &= (P-1) * T_{\text{exch}}(n/P) \\
&= (P-1) * (2L + 2o + (\left\lceil \frac{(n/P)}{w} \right\rceil - 1 - L/g) \max(g, 2o))
\end{aligned}
$$

One source of stalls under LogP occurs when the capacity limit of the network is exceeded. We can ensure that no more than L/g messages are in transit to any processor by constructing a communication schedule such that no two processors store into the same processor on the same iteration. This is achieved by having each processor p store to processor $p \oplus i$, where i is the current iteration. With this schedule only pairs of processors exchange data, if we assume the processors operate synchronously.[5]

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 in each iteration of the remap, messages may not be retrieved from the network in 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.[6] We would like to point out that

[5] The astute reader may note 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 misses and network collisions, this approach increases the likelihood that there will be contention.

[6] The model used in this comparison and in the graphs to follow was adjusted to include the overhead

our first several attempts to implement the remap failed to completely address the potential stalls articulated in the model and the performance suffered. Failing to meet the model predictions motivated us to look more closely at the implementation and the model served further to explain the defect.

1.5.4 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. Analyzing the behavior of bitonic sort gives us the following equation

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

The equation for the communication phase, T_{remap}, was given in Section 1.5.3. The models for the local computations consist of the measured time per key for that computation, as designated in Table 1.1, multiplied by the number of keys and the number of times the computation is performed.

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

1.5.5 Empirical Results

Figure 1.4 shows the predicted and measured time per key for bitonic sort between 16K and 1M keys per processor on a CM-5 of 32 to 512 processors. The time per key per increases only slightly with increasing 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 very close to the measured time; all of our errors are less than 12%.

These figures can also be used to derive the speedup for a constant problem size on an increasing number of processors. For example, sorting a total of 32M keys on 32 processors requires time $34.6\mu s * 1M = 36.3$ seconds; sorting the same number of keys on 512 processors requires time $61.5\mu s * 64K = 4.0$ seconds. This decrease in execution

of Split-C's implementation of bulk stores. An additional cost of $2L + 4o$ is incurred for each invocation of bulk store.

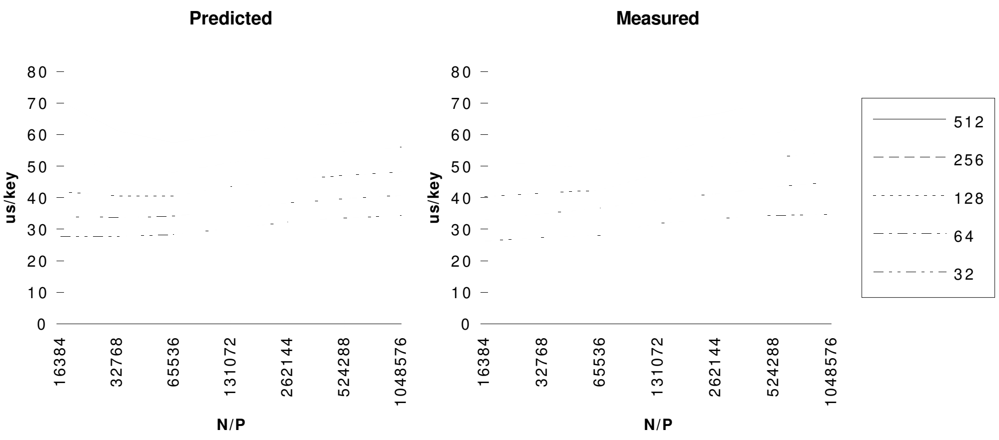

Figure 1.4 *Predicted and measured execution time per key of bitonic sort on the CM-5. Times are shown to sort between 16K and 1M keys per processor on 32, 64, 128, 256 and 512 processors.*

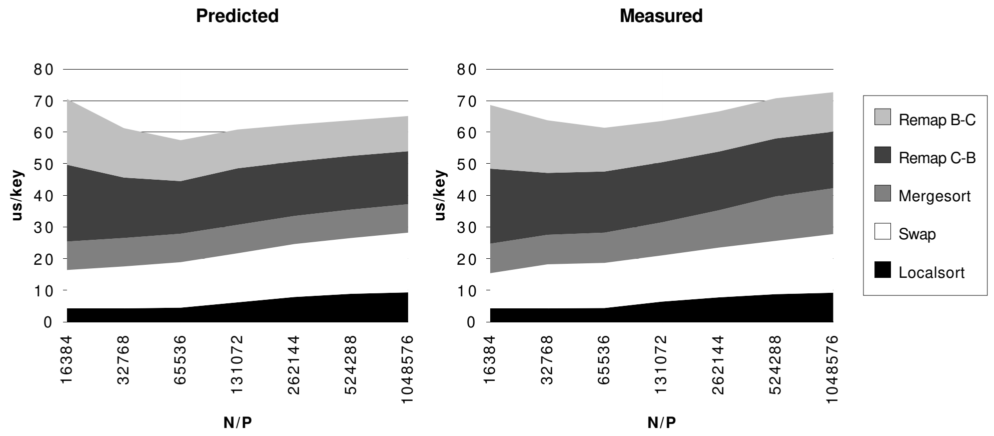

Figure 1.5 *Predicted and measured execution times per key on 512 processors for the phases of bitonic sort. The time for the single gather is included in the time for the remap from a blocked to a cyclic layout; likewise, the time for the scatter is included in the time for the remap from a cyclic to a blocked layout.*

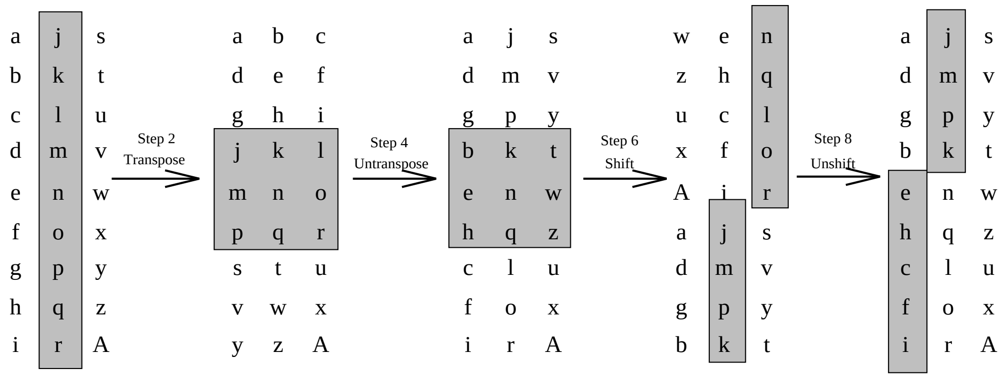

Figure 1.6 *The permutation operations performed in column sort on a 9 x 3 matrix. The shaded regions show the positions to which processor 1 moves its keys.*

time corresponds to a nine-fold speedup, with a 16-fold increase in the number of processors. This drop in efficiency is illustrated in the figure by the increasing time per key per processor as the number of processors is increased.

Figure 1.5 shows the breakdown of the execution time of bitonic sort into 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. 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. Our prediction of the time for the remap is always within 12% of the measured execution time. It is worthwhile to note that our predictions for the communication phases and for the local computation have roughly the same accuracy; however, most of the inaccuracy in modeling the local computation is in the bitonic merge sort which we model with only a constant time per key.

1.6 COLUMN SORT

Column sort [Lei85], 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, as illustrated by Figure 1.6 for $n = 9$ and $P = 3$. A local sort is performed on every column before each communication phase. Transpose is exactly the blocked-to-cyclic remap described in bitonic sort. The keys transposed from one processor to another are in contiguous locations on the destination processor, but are in locations separated by a constant stride in the source processor, so a gather is performed on the

Variable	Operation	Time Per Key (μs/key)	
t_{merge}	merge two sorted lists of keys	1.5	
$t_{\text{shiftcopy}}$	shift key	0.5	
t_{gather}	move key before blocked to cyclic remap	0.52 1.1	if $n \leq$ 64K or $P \leq 64$ otherwise
$t_{\text{localsort}}$	local radix sort of keys	4.1 $-22.7 + 1.7 * \lg n$ $-0.7 + 0.5 * \lg n$	if $n <$ 64K if 64K $\leq n \leq$ 256K if 256K $< n$

Table 1.2 *Models of local computation times per key in column sort.*

source. Untranspose is the cyclic-to-blocked remap, so keys in contiguous locations on the source processor are untransposed to locations that are scattered on the destination processor. However, the scatter can be avoided because the next step in column sort is a local sort on the column.

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 only one-to-one communication. A small amount of local computation is also performed to move the first half of the keys on each processor to second half of the same column. Similarly, in the unshift operation, the first half of the keys on processor i is sent to the second half of column $(i-1) \bmod P$. The local computation consists of moving the second half of each column to the first half.

Optimizing the local computation in column sort is easier than in bitonic sort, because the only local computation is a local sort. The important issue is that after the first sort, we need only sort partially sorted data. In the second and third local sorts, there are P sorted lists of length n/P on each processor, so a P-way merge could be performed instead of a general sort. However, empirical study showed that our general local radix sort was faster than a P-way merge sort for the P of interest. In the last sorting step there are two sorted lists of length $n/2$ in each column and a two-way merge is most efficient. The model for this local computation is found in Table 1.2.

1.6.1 LogP Complexity

The execution time of column sort is the sum of its eight steps.

$$\begin{aligned} T_{\text{columnsort}} &= T_{\text{localsort}} + T_{\text{transpose}} + T_{\text{localsort}} + T_{\text{untranspose}} \\ &\quad + T_{\text{localsort}} + T_{\text{shift}} + T_{\text{merge}} + T_{\text{unshift}} \end{aligned}$$

The time of the local sort, $T_{\text{localsort}}$ is the same as in bitonic sort. Our implementation of transpose gathers the keys into contiguous memory before storing them to any of the destination processors. The cost of the transpose is represented as follows.

$$T_{\text{transpose}} = T_{\text{gather}} + T_{\text{remap}}$$

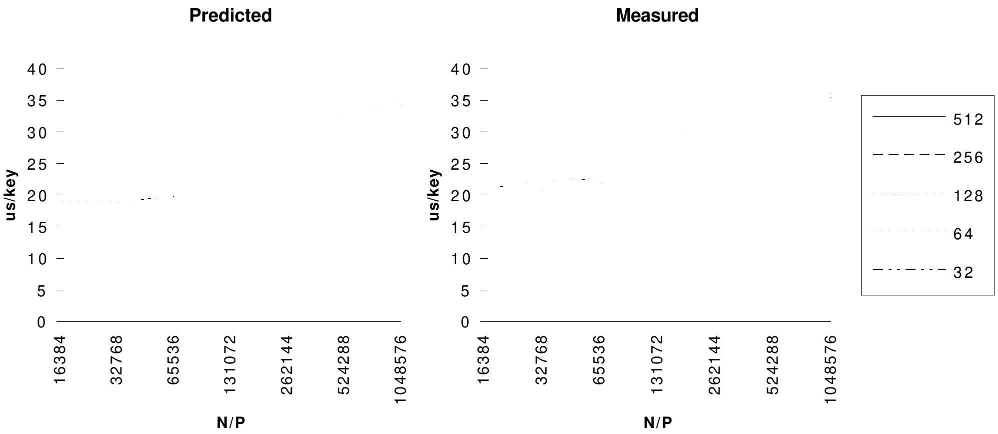

Figure 1.7 *Estimated and measured execution time of column sort on the CM-5. Note that, due to the restriction that $N \geq P^3$, our algorithm cannot sort all values of n for a given P.*

Untranspose does not require a scatter; therefore, it is modeled as a single remap.

$$T_{\text{untranspose}} = T_{\text{remap}}$$

The shift and unshift consist of a local copy, receiving $n/2$ words from one processor, and sending the same amount to another. This has the same cost as a pairwise exchange. Since n is large compared to L, we may simplify the model to the following.

$$T_{\text{shift}} = T_{\text{unshift}} = \frac{n}{2} * t_{\text{shiftcopy}} + \left\lceil \frac{n}{2w} \right\rceil * \max(g, 2o)$$

Finally, the time for the 2-way merge is

$$T_{\text{merge}} = n * t_{\text{merge}}$$

1.6.2 Empirical Results

The predicted and measured time per key for column sort is shown in Figure 1.7. 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 slightly more. The small increase is due to the rise in gather time with the number of processors. 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, whose time per key increased significantly with P, but only very slowly with n. Due to these constraints, column sort is faster than bitonic sort for small data sets on a small number of processors, but is slightly slower for large data sets on a small number of processors; for a large number of processors, column sort is once again faster.

Figure 1.8 shows the predicted and measured times per key for 64 processors. This

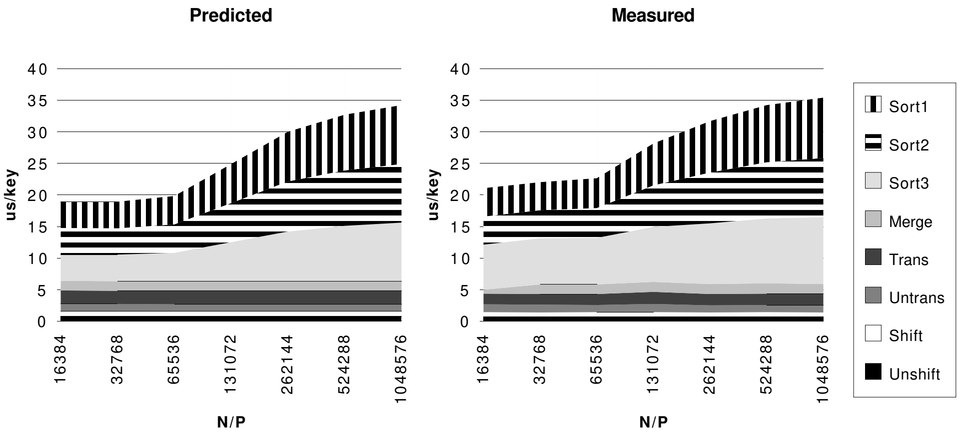

Figure 1.8 *Estimated and measured execution time of the phases of column sort on 64 processors.*

configuration was chosen because it is the largest number of processors on which the full range of keys can be sorted. These 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 for the communication phases remain 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 in our prediction is for the local computation.

1.7 RADIX SORT

Parallel radix sort requires fewer local computation and key distribution phases than our 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 itself involves local computation and global communication, namely a multi-scan and a multi-broadcast.

The parallel version of radix sort is a straight-forward extension to a local radix sort. Recall, radix sort relies on the representation of keys as b-bit numbers. Each digit in the key has r bits, where r is defined to be the *radix*. Radix sort performs $\lceil b/r \rceil$ passes over the keys, each time determining the rank of each digit and permuting the keys to their destinations. In the parallel version, the keys are in a blocked layout across processors. The destination address of each key is obtained by computing a global histogram and the permutation involves all-to-all communication. Specifically, each pass of the parallel radix sort consists of three phases. First, each processor computes a local histogram containing 2^r buckets by traversing its list of keys and counting the number of occurrences of each of the 2^r digits. Second, the global rank of each key is calculated from the local histograms such that the global histogram 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. At this point,

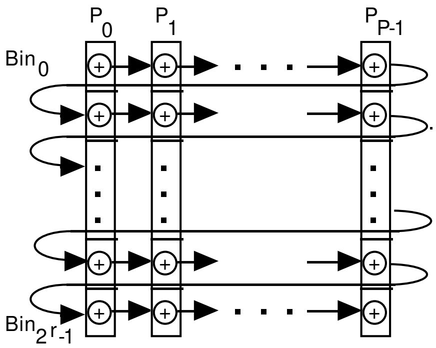

Figure 1.9 *Linear scan of local histograms to form the global histogram.*

each processor has the global rank of the first of its keys for each digit, which is then used to distribute each key to its proper position.

In the naive approach to constructing the global histogram, each row of the global histogram depends upon the previous row.

$$\text{GlobalHist}[p][i] = \begin{cases} \sum_{j=0}^{p-1} \text{LocalHist}[j][0] & \text{if } i = 0 \\ \text{GlobalHist}[P-1][i-1] + \sum_{j=0}^{p-1} \text{LocalHist}[j][i] & \text{otherwise} \end{cases}$$

This construction is shown in Figure 1.9. When this dependency exists, a parallel prefix could be used for each bin, but the 2^r scans would be done sequentially. For a large radix, this is inefficient. To eliminate the dependency, a *partial global histogram* is constructed, whose contents are

$$\text{PartialHist}[p][i] = \sum_{j=0}^{p-1} \text{LocalHist}[j][i]$$

Each row in the partial histogram is now independent of the earlier rows. After the partial global histogram is constructed, the sum of each row of buckets, stored in PartialHist$[P-1][i]$, where $0 \le i < 2^r$, is broadcast to all processors. Each processor computes its portion of the global histogram by adding the sum of all lower buckets in the histogram that was broadcast to the bucket 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, a key with digit value k on processor p is stored in the (GlobalHist$[p][k] \bmod n$)-th position on processor GlobalHist$[p][k]$ div n. In the straight-forward implementation of this step, each processor loops through its set of keys, determines the destination processor and offset of each key, and stores each key on the correct position. This leads to all-to-all communication which is irregular, yet balanced. In irregular communication, the destination of keys is dependent upon the value of the key; therefore, it is not possible to precompute a contention-free communication schedule and the analysis is more challenging.

1.7.1 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 $2ng$. Thus, if n is much larger than P, it is faster to broadcast the data as a linear pipeline, than as a tree.[7] 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. Multi-broadcast is identical to multi-scan, except the value is forwarded directly to the next processor without adding it to a local value, so we concentrate on the multi-scan for the remainder of the discussion.

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

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

However, we are tacitly assuming in this analysis 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 the 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 case is that processor 0 is only transmitting, so it can send values faster than the rest of the pipeline can forward them. If we are not careful, processor 1 may receive multiple values before forwarding previous ones, and so on for the remaining processors.

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

[7] In [Karp93] 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.

Variable	Operation	Time Per Key or Bucket (μs/key or μs/bin)
t_{zero}	clear histogram bin	0.22
t_{h}	produce histogram for key	1.2
t_{add}	produce scan value to forward	1.0
t_{bsum}	adjust scan of bins	2.5
t_{addr}	determine destination of key	4.7

Table 1.3 *Models of local computation in radix sort.*

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. Although it is well known that greedy scheduling can be suboptimal, it seemed counter-intuitive that slowing down one processor would make the whole operation proceed much faster. Without the model as a guide, we would probably not have noticed the deficiency in the implementation, much less understood that this was a reasonable way to correct it.

1.7.2 Irregular Balanced Communication

The key distribution phase in radix sort may involve an arbitrary permutation of the data, so there is no fixed contention-free communication schedule, as in the remaps in bitonic and column sort. We could transform this irregular communication pattern into regular communication by first gathering the keys destined for each other processor and then performing a series of pairwise exchanges. However, each processor may send a different number of keys to every other processor. Alternatively, we can simply tolerate the contention that occurs at the destinations. Determining the expected slow-down due to contention of a random permutation under LogP is an interesting open problem. Our simulations and recent theoretical results [Liu93] 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. For radix sort, an address calculation is performed before storing each element, and this local computation time further mitigates the effects of contention. Thus, we elected to ignore the contention and model the distribute phase simply as

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

where t_{addr} denotes the time to access a key and compute the destination address.

1.7.3 LogP Complexity

In this section, we summarize the predicted execution time of radix sort. 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.

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

$T_{\text{localhist}}$ involves only local computation; the time to form the local histogram is simply the time to initialize each of the histogram buckets and the time to increment the corresponding bucket for each of the keys

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

The models of t_{zero} and t_{h} along with the other local computations in radix sort, are presented in Table 1.3. The LogP complexity of constructing the global histogram, is the sum of three components

$$T_{\text{globalhist}} = T_{\text{scan}} + T_{\text{bcast}} + T_{\text{final}}.$$

Our models of the execution time of T_{scan} and T_{bcast} are those presented in Section 1.7.1, plus the cost of initializing each of the histogram buckets

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

Finally, the keys are permuted to their destinations in time

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

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, with a larger radix, less passes are performed. In our implementation we use a radix size of 16 bits.

1.7.4 Empirical Results

In Figure 1.10, the predicted and measured times per key are plotted for radix sort. For small values of n, our measurements are only 9% higher than the prediction; for large values of n, our measurements are 37% higher. Our prediction from the model is that increasing the number of processors has a negligible impact on the execution time per key. Our 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 sort, it is faster for large n.

Figure 1.11 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 nThe model matches the measured execution time precisely for 32 processors and 16K keys. On 512 processors and 16K keys, our measured time is 34%

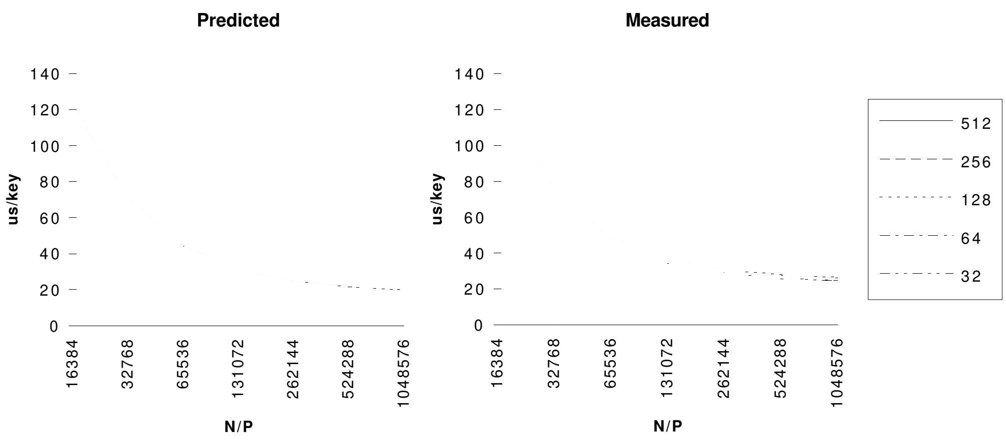

Figure 1.10 *Predicted and measured execution time per key of radix sort on the CM-5.*

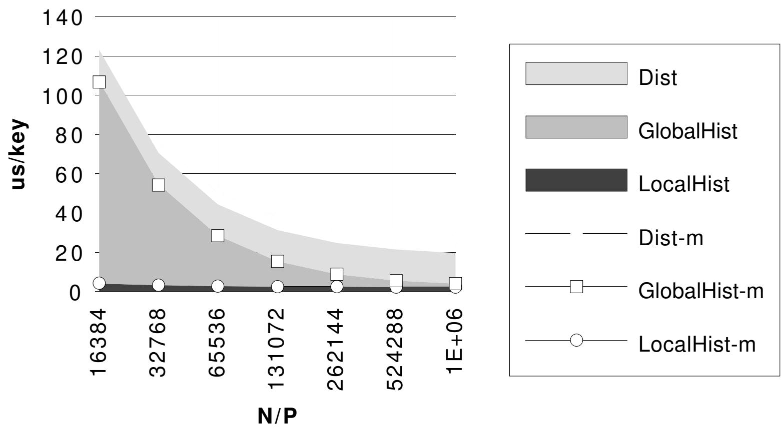

Figure 1.11 *Predicted and measured execution times per key of various phases in radix sort on 512 processors.*

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 an indication of contention which we do not model. However, we believe that the increase is largely due to cache effects because if keys are distributed to the correct destination processor, but always to the same memory location instead of to the correct offset determined by the keys' rank, the time per key does not increase with n. 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.

1.8 SAMPLE SORT

An interesting recent algorithm, called sample (or splitter) sort [Ble91], 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 can be 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 then 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. (A key with value k is sent to the first processor p such that splitter$[p] \leq k \leq$ splitter$[p+1]$.) In the last phase, a local radix sort is performed on the received keys.

Note that each processor potentially receives a different number of keys, so the communication phase is *unbalanced*. We call the ratio of the maximum number of keys received by a processor to the average the *expansion factor*. 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 [Ble91].

The local sort time in sample sort is different than that in bitonic and column sort for two reasons. First, the number of keys being sorted may be as large as $E*n$, instead of n, where E is the expansion factor, and the rate per key is dependent on $E*n$, and $T_{\text{localsort}} = E*n*t_{\text{localsort}}$. Secondly, because each processor sorts only keys whose values fall between two splitters, the values of the keys cover a much smaller range than the keys in bitonic or column sort. Due to caching effects, the local radix sort is faster under this condition and we have a different piece-wise model for the local radix sort. The model for this and the other local computations in sample sort is given in Table 1.4.

Variable	Operation	Time Per Key (μs/key)	
t_{compare}	compare key to splitter	0.9	
$t_{\text{localsort\_8}}$	local radix sort of samples	5.0	
$t_{\text{localsort}}$	local radix sort of keys	4.1	if $E * n <$ 64K
		$-14.3 + 1.2 * \lg(E * n)$	if 64K $\leq E * n \leq$ 256K
		$-0.6 + 0.4 * \lg(E * n)$	if 256K $< E * n$

Table 1.4 *Models of local computation times in sample sort.*

The key distribution step of sample sort involves all-to-all communication which is neither balanced nor regular. 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, but the local computation per key is more expensive, since a binary search is involved. Thus, we continue to ignore the potential contention. However, the keys are not distributed evenly to destination processors; one processor may receive up to $E * n$ keys. The execution time of this step is limited by the processor receiving the most keys.

1.8.1 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 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 1.7.1 for P values.

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

As described in Section 1.8.1, the time for the key distribution phase is

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

We model the time of the binary search on the splitter array for each key as the number of compares multiplied by the time to perform a single compare operation

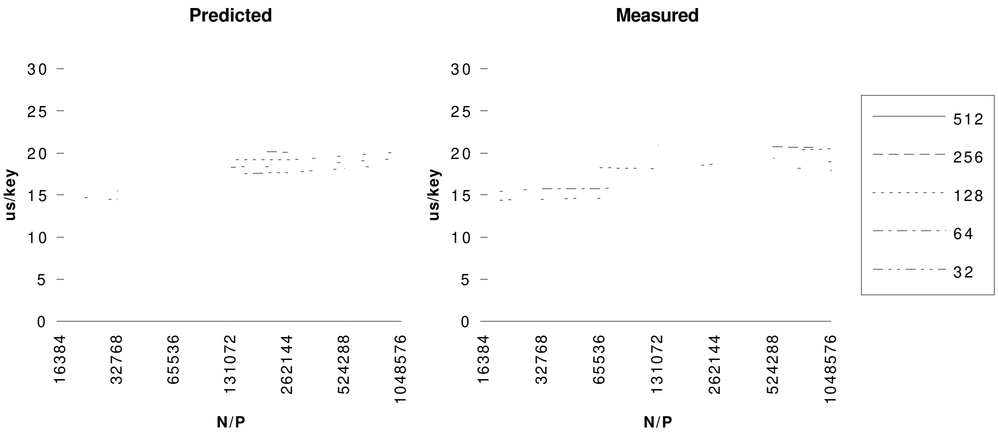

Figure 1.12 *Estimated and measured execution time of parallel sample sort on the CM-5.*

$$t_{\text{search}} = \lg P * t_{\text{compare}}$$

For our sample size, number of keys per processor, number of processors, and distribution of 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.

And, as described previously,

$$T_{\text{localsort}} = E * n * t_{\text{localsort}}$$

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.

1.8.2 Empirical Results

The predicted and measured times per key for sample sort are displayed in Figure 1.12. 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 discusses 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. Sample sort is faster than all three previous sorts for all measured data points. The measured execution times of sample sort exhibit variation with n because the actual expansion factors are dependent on the random data sets; we observed expansion factors varying between 1.2 and 1.4.

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

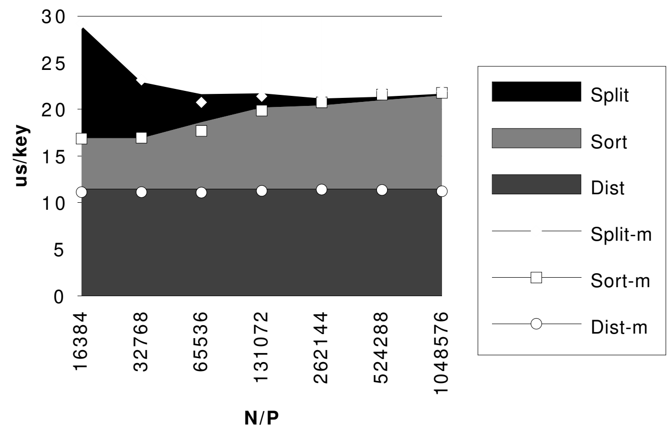

Figure 1.13 *Estimated and measured execution times of various phase of parallel sample sort on 512 processors.*

Figure 1.13. Each of the three phases in sample sort behaves differently as the number of keys per processor varies. The 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 an oversampling ratio of $s = 32$ instead of 64[8].

1.9 SUMMARY

The LogP model characterizes the performance of parallel machines with a small set of parameters: latency (L), overhead (o), bandwidth (g), and the number of processors (P). The LogP model does not specify how local computation is modeled; we chose to perform our predictions based on measurements of the execution time. In this paper we have shown that LogP not only accurately predicts the communication phases of fast parallel sorts on the CM-5, but also exposes deficiencies in implementations.

We found that when parallel sorting algorithms are highly optimized to exploit the layout of keys across processors the algorithms consist of alternating phases of local computation, key distribution, and setup to determine the destination of keys. The lower the number of communication phases performed in a sort, the more complex are those phases and their setup.

The accuracy of the predictions of the model encourages us to extrapolate to config-

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

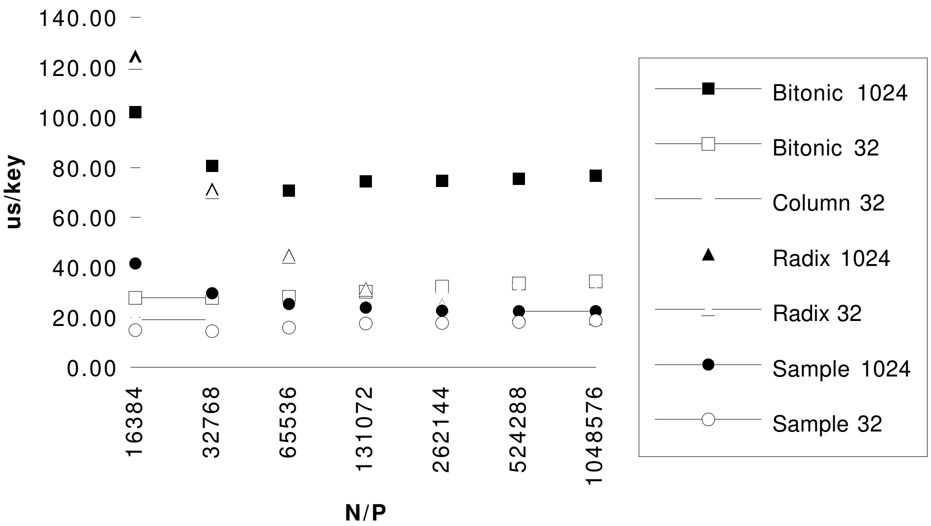

Figure 1.14 *Estimated execution time of four parallel sorting algorithms under LogP with the performance characteristics of the CM-5.*

urations which we have not yet measured. Figure 1.14 shows the predicted execution time of the four sorts 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 meet our goal of sorting one billion keys in less than a minute. Actually, both sorts can solve the problem 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 1.7.4, 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. Radix sort has the advantage that its execution time is less sensitive to the input distribution of keys than sample sort; the execution time of sample sort is dependent upon the expansion factor, which may vary greatly with different key distributions. All of our experiments were performed on random distributions of keys and thus have relatively low, predictable expansion factors.

Another question of interest when considering the merits of a sorting algorithm is the execution time given a certain problem size and number of processors. Let us look at four operating regimes. As discussed above, for a large number of processors and large problem size, the performance of radix and 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. 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; however, the expansion factors in sample sort may vary more widely than with larger numbers of processors. Thus, radix sort should be used for this range of problems. Finally, with small data sets on a large number of processors, sample sort significantly outperforms the other sorts.

Future work could investigate several interesting optimizations on the parallel sorts. As our figures illustrates, both radix and bitonic sort have a substantial startup cost for small data sets. For radix sort, instead of using a constant radix size of 16, the radix could be varied depending upon the operating regime. When sorting a small number of keys, a smaller radix size would undoubtedly improve performance. Similar improvements could also be achieved in sample sort by using a smaller oversampling rate for small data sets and large number of processors. We have confidence in the predictions of the LogP model so that the merits of these optimizations could be determined without actually being implemented and measured.

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. 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

[Akl89] S. Akl. The Design and Analysis of Parallel Algorithms. Prentice Hall, 1989.

[Bat68] K. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, Vol. 32, 1968.

[Ble91] 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.

[Cul93a] 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. Also appears as UCB/CS/92 713 report.

[Cul93b] 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 93.*

[Karp93] R. Karp, A. Sahay, and E. Santos. Optimal Broadcast and Summation in the LogP Model, In *5th Symp. on Parallel Algorithms and Architectures, June 1993.*

[Len92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, The Stanford Dash Multiprocessor *IEEE Computer*, 25(3):63-79, March 1992.

[Lei85] T. Leighton. Tight bounds on the comlexity of parallel sorting. In *IEEE Transactions on Computers*, April 1985.

[Lei92] C. E. Leiserson, et al. The Network Architecture of the Connection Machine CM-5 In *Symposium on Parallel Algorithms and Architectures*, April 1992.

[Liu93] P. Liu, W. Aiello, S. Bhatt. An atomic model for message-passing. In *Symposium on Parallel Algorithms and Architectures*, June 1993.

[Rei87] J. H. Reif and L. G Valiant. A Logarithmic time Sort for Linear Size Networks, *Journal of the ACM,* 34(1):60-76, January, 1987.

[Sha49] C. Shannon and W. Weaver. The Mathematical Theory of Communication, University of Illinois Press: Urbana, 1949.

[The91] K. Thearling and S. Smith. An improved supercomputer sorting benchmark. Thinking Machines Corporation, 1991.

[VEi92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *International Symposium on Computer Architecture*, 1992.

[Zag91] M. Zagha and G. Blelloch. Radix Sort for Vector Multiprocessors, In *Supercomputing 91.*