Radix Sort For Vector Multiprocessors*

Marco Zagha and Guy E. Blelloch
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

We have designed a radix sort algorithm for vector multiprocessors and have implemented the algorithm on the CRAY Y-MP. *On one processor of the* Y-MP, *our sort is over 5 times faster on large sorting problems than the optimized library sort provided by* CRAY *Research. On eight processors we achieve an additional speedup of almost 5, yielding a routine over 25 times faster than the library sort. Using this multiprocessor version, we can sort at a rate of 15 million 64-bit keys per second.*

Our sorting algorithm is adapted from a data-parallel algorithm previously designed for a highly parallel Single Instruction Multiple Data (SIMD) computer, the Connection Machine CM-2. To develop our version we introduce three general techniques for mapping data-parallel algorithms onto vector multiprocessors. These techniques allow us to fully vectorize and parallelize the algorithm. The paper also derives equations that model the performance of our algorithm on the Y-MP. *These equations are then used to optimize the radix size.*

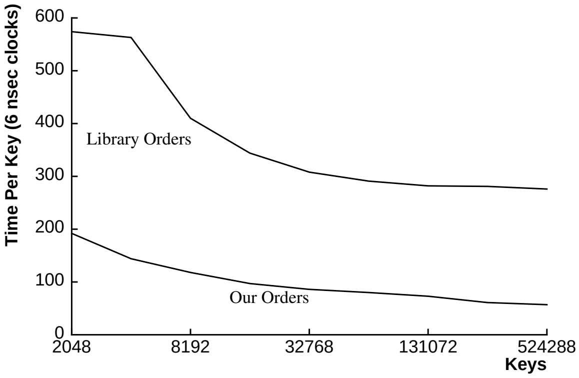


Figure 1: Comparison of CRAY library ORDERS and our implementation of ORDERS on 64-bit random keys using one processor. ORDERS is a version of sort that returns the positions of the sorted keys rather than the actual sorted keys. Our implementation of SORT is about 10% faster than our implementation of ORDERS.

1 Introduction

Sorting is one of the most heavily studied problems in computer science. A sorting algorithm, bitonic sort [1], was one of the first algorithms designed for parallel machines, and since then hundreds of articles on parallel sorting have appeared in the literature. Although this work has developed a robust theory of parallel sorting, there has only been limited success in obtaining efficient implementations on real parallel machines. Many of the reported results are barely faster than serial sorting algorithms on fast workstations. This is due to the high communication costs of sorting, the large constant factors hidden by asymptotic analysis [15], and the fact that although the theoretical sorts are efficient in some theoretical models of a parallel machine, these models do not accurately portray real machines. Many of the sorting algorithms are thus rendered impractical.

This paper discusses a practical radix sort designed for vector multiprocessors and implemented on an 8 processor CRAY Y-MP. The sort fully utilizes both the vector and multiprocessor facilities of the Y-MP, and for large data sets achieves a running time that is over 200 times faster than an optimized sort on a fast workstation, and 25 times faster than the highly optimized library sort supplied by CRAY Research for a single processor of the Y-MP [8]. Figure 1 compares the CRAY library sort and our sort for a single processor of the Y-MP (CRAY Research does not supply a multiprocessor version of the sort). The sort was implemented on the Y-MP using our own macro assembler to generate Cray Assembly Language (CAL) [9].

Our sorting algorithm is adapted from a data-parallel radix-sort algorithm previously designed for the Connection Machine CM-2 [3]. To generate an efficient vector-

*This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. Cray Y-MP time was provided by the Pittsburgh Supercomputing Center (Grant ACR552P).

multiprocessor algorithm from the data-parallel algorithm, we use the following techniques:

Virtual Processors: Each element of a vector register is viewed as a *virtual processor*. For a vector-register length L on each processor, and for P processors, we therefore use an algorithm designed for an $L \times P$ processor machine ($64 \times 8 = 512$ on the Y-MP). Viewing each vector element as its own processor forces independence among vector elements, allowing the algorithm to be fully vectorized.

Loop Raking: A new technique we call *loop raking* is used for vectorizing the loops. Instead of loading contiguous blocks of a vector on each vector load, as done in strip mining [16], loop-raking uses a constant stride to load a set of elements evenly distributed across the input vector—as if a rake was placed over the vector. Loop raking is necessary to keep each pass of the radix-sort stable (see Section 3).

Processor Memory Layout: A careful layout of the memory of each virtual processor is used to avoid bank conflicts. This layout is used for placing the buckets needed by the radix sort and guarantees that for every vector gather from or scatter to the buckets, each element of a vector register accesses a different memory bank. As compared to a naive layout of the buckets in contiguous memory locations, our layout greatly reduces the running time for some common inputs.

In addition to describing the algorithm, this paper models the running time of the radix sort in terms of the number of keys to be sorted, the size of each key, and the number of processors. The model yields the following equation for the number of 6 nsec clock cycles:

$$T = \frac{b}{r}(230 \cdot 2^r + 7.1\frac{N}{P})/(1 - .22(P/P_m)^{1.5}) \quad (1)$$

where N is the number of keys, P is the number of physical processors used, P_m is the maximun number of physical processors (8 on the Y-MP), r is the number of bits in the radix, and b is the number of bits in the key. The form $(b/r)(k_1 2^r + k_2(N/P))$ is based on the theoretical complexity, and the constants k_1 and k_2 are derived from empirical measurements on the Y-MP. The $(1 - .22(P/P_m)^{1.5})$ term is due to memory contention and is also based on empirical measurements. Equation 1 is used to select an optimal radix size r (typically between 6 and 14), and can also be used to predict running times over a range of data sizes and numbers of processors without having to run the algorithm. With $N > 5000$, equation 1 predicts the actual running time within 10% accuracy. The inaccuracy is due to a slight dependence of the running time on the distribution of the keys.

The remainder of this paper is organized as follows. Section 2 discusses the serial and parallel versions of radix sort. Section 3 discusses the use of virtual processors, loop raking, and memory layout. Section 4 describes our implementation on the CRAY Y-MP. Section 5 evaluates the performance of our implementation and discusses how to select the radix size.

2 Radix Sorting

Radix-based sorting algorithms treat keys as multidigit numbers in which each digit is an integer with a value in the range $\langle 0 \ldots (m-1) \rangle$, where m is the radix. A 32-bit integer, for example, could be treated as a 4-digit number with radix $m = 2^{32/4} = 2^8 = 256$. The radix m is usually chosen to minimize the running time and is highly dependent on the implementation and the number of keys being sorted. The standard radix sort on which our parallel algorithm is based is perhaps the oldest form of sorting. The first implementation dates back to the 19th century and Hollerith's machines for sorting punched cards.

Radix sorting is an attractive alternative to comparison-based sorts, such as quicksort [10], since for n keys it runs in $O(n)$ instead of $O(n \lg n)$ time. In practice, however, a disadvantage is that it accesses memory in an arbitrary order. Therefore, on a machine with a cache, many memory references will cause cache misses. Studies have shown that on serial machines when the problem fits into physical memory but not into the cache, radix sort performs no faster, or only marginally faster, than optimized implementations of quicksort (for 32-bit and 64-bit integers) [14, Chapter 8].

Since the Y-MP has no cache—all physical memory is accessible with equal cost—and assuming radix sort can be vectorized, radix sort is likely to have better relative performance over comparison-based sorts than on cache based machines. This seems to be true. For large data sets, our single processor sort runs between 3 and 5 times faster than a fully vectorized version of quicksort implemented by Levin [13].[1]

Serial Radix Sort

We first review the serial radix sort since our parallel algorithm will include the same phases (see [6, Section 9.3] for more details). Radix sort works by breaking keys into digits and sorting one digit at a time, starting with the *least* significant digit. Figure 2 shows an example. A *counting*

[1] This takes into account the difference in clock rates between the X-MP and Y-MP.

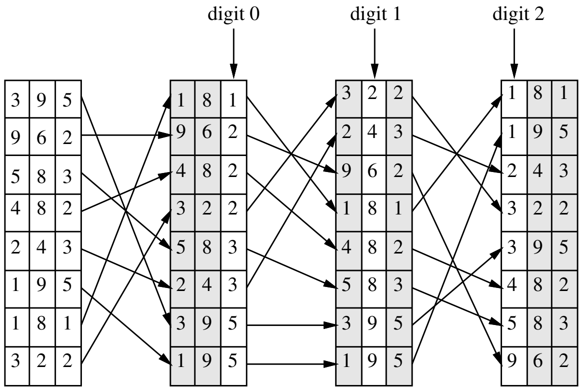


Figure 2: Radix Sorting 3-digit numbers 1 digit at a time. A counting sort is used for each digit.

sort (sometimes called distribution sort) is used to sort each digit. Since RADIX-SORT starts from the least significant digit, the sort works only if the ordering generated in previous passes is preserved. Each COUNTING-SORT therefore must be stable.

The COUNTING-SORT is implemented as follows. We assume that each digit consists of r bits: $m = 2^r$. The basic idea is to determine, for each input digit $D[i]$, the number of keys with a digit less than $D[i]$, and also the number of keys with a digit equal to $D[i]$ appearing earlier in the input sequence. To do this, the sort uses m buckets, one for each possible digit value. The following routine sorts an array of keys K, based on an array of digits D, both of size N, r bits at a time, placing the result in R. The algorithm is divided into 3 phases:

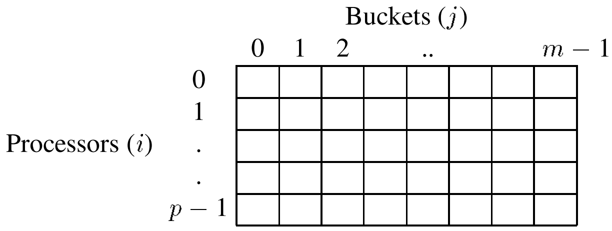
```
COUNTING-SORT
  HISTOGRAM-KEYS
    do i ← 0 to 2^r − 1
      Bucket[i] ← 0
    do j ← 0 to N − 1
      Bucket[D[j]] ← Bucket[D[j]] + 1
  SCAN-BUCKETS
    Sum ← 0
    do i ← 0 to 2^r − 1
      Val ← Bucket[i]
      Bucket[i] ← Sum
      Sum ← Sum + Val
  RANK-AND-PERMUTE
    do j ← 0 to N − 1
      A ← Bucket[D[j]]
      R[A] ← K[j]
      Bucket[D[j]] ← A + 1
```

The first loop of HISTOGRAM-KEYS clears the buckets. The second loop, at iteration j, uses the j^{th} element of the digit array as an offset into the buckets, and increments the count in that bucket. At the end of HISTOGRAM-KEYS, $Bucket[i]$ contains the number of digits having value i. SCAN-BUCKETS performs a *scan* (also called all-prefix-sums) operation on the buckets, returning to each element the sum of all previous elements. After the scan, $Bucket[i]$ contains the number of digits with a value j such that $j < i$. This is the position in the output in which the first key with digit i belongs. In the final phase, RANK-AND-PERMUTE, each key with a digit of value i is placed in its final location by getting the offset from $Bucket[i]$ and incrementing the bucket so the next key with digit i gets put in the next location. Since the keys are looped over in increasing order, COUNTING-SORT is stable.

Parallel Radix Sort

The serial algorithm cannot be directly parallelized (or vectorized) because of loop dependencies in all three phases. If an attempt is made to parallelize over the iterations of the HISTOGRAM-KEYS, several processors could try to increment the same bucket simultaneously. If the bucket is not locked by each processor to gain exclusive access, only one of the increments would take effect. On the other hand, if the bucket is locked, the bucket would act as a serial bottleneck and would greatly degrade performance if many keys had the same digit.

By using a separate set of buckets for each processor, it is possible to parallelize the algorithm without requiring a lock on the buckets. Each processor is responsible for its own N/P subset of the keys and tabulates them in its own set of buckets. A restricted version of this algorithm was described by Cole and Vishkin as part of an optimal 2-ruling set algorithm [5]. The full algorithm was also implemented on the Connection Machine CM-2 [3]. In this parallel version of radix sort, the buckets can be viewed as a matrix $Buckets[i, j]$:



where i is the processor number and j is the bucket number. Now in the first and third phases of radix sort (HISTOGRAM-KEYS and RANK-AND-PERMUTE) each processor works on its own set of keys with its own set of buckets, thereby removing all dependencies. SCAN-BUCKETS, however, must be modified to merge the buckets from the different processors. If each processor only scanned its own buckets, the digits would be sorted within the processor but not across the whole input data. After SCAN-BUCKETS, we would like $Buckets[i,j]$ to contain the position in the output where the first key from processor i with digit j belongs (see Figure 3). This can be expressed as:

$$Buckets[i,j]_{\text{after}} = \sum_{k=0}^{p-1}\sum_{m=0}^{j-1} Buckets[k,m] + \sum_{k=0}^{i-1} Buckets[k,j] \ .$$

That is to say, the offset is the total number of digits less than j over all the processors ($0 \leq k < p$), plus the number of digits equal to j in processors less than i. This sum can be calculated by flattening the matrix into column major order and executing a SCAN-BUCKETS on the flattened matrix. The SCAN-BUCKETS operation can be parallelized using a tree-summing or similar algorithm [12]. The parallel version generates the same permutation as the serial algorithm so the sort remains stable.

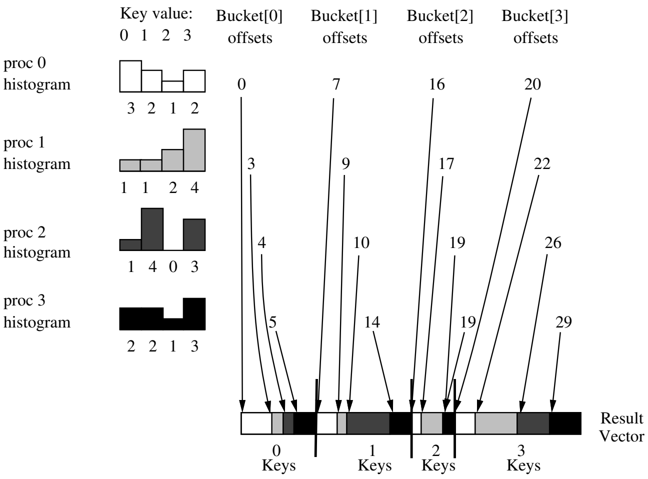


Figure 3: The scan step of parallel radix sort. The algorithm is illustrated with 4 processors and 4 buckets for the values 0–3. The offsets are computed by scanning the buckets. After the scan, each processor has the starting position in the final output of keys with a particular value. For example, processor 3 will place the "0" keys starting at position 5 in the output, and the "1" keys starting at position 14, as indicated by offsets.

3 Algorithmic Techniques

The previous section described a data-parallel radix-sort algorithm. This section describes three general techniques that can used to map this algorithm onto vector mutiprocessors: *virtual processors*, *loop raking*, and *processor memory layout*. We have also used these techniques in the implementation of a variety of other parallel algorithms on the CRAY Y-MP and Ardent Titan.

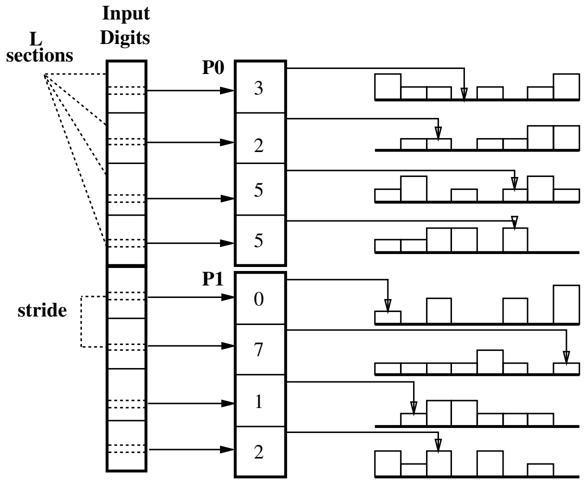


Figure 4: Vector Multiprocessor Histograming. Each processor has a set of buckets for each vector register element (virtual processor). The figure depicts 2 processors ($P = 2$) using a vector length of 4 ($L = 4$). For a 3-bit digit ($r = 3$), there are $2^r \cdot P \cdot L = 8 \cdot 2 \cdot 4 = 64$ buckets. Using P processors, the keys are divided into P equal sections, which are further divided into L sections, one section for each virtual processor.

Virtual Processors: Vector multiprocessors, such as the CRAY, Convex, or Ardent Titan, offer two levels of parallelism: multiprocessor facilities and vector facilities. To take full advantage of these machines it is important to use both levels. One way is to flatten the two levels into one homogeneous level by viewing each element of a vector-register as a *virtual processor*. We call these elements virtual processors so as not to confuse them with a full vector processor. In this view, a machine with a vector-register length L, and with P processors, will have $L \times P$ virtual processors. Since the Y-MP has a maximum vector-register length of 64, and there are up to 8 processors, the CRAY can contain up to 512 virtual processors. Section 5 discusses why using a smaller vector length, and hence number of virtual processors, can improve performance in certain cases. On machines with more flexible use of the vector registers, such as the Ardent Titan, it is up to the algorithm designer to decide how many virtual processors to use. This decision should be based on the vector startup times, the number of vector registers needed, and the cost of using additional virtual processors.

Any data-parallel algorithm can be both vectorized and parallelized by assigning each processor in the algorithm to a virtual processor on the machine. In our implementation of the parallel radix sort on the CRAY Y-MP, the keys are divided into equally sized sets, and each virtual processor is responsible for one of these sets. Each virtual processor also uses its own set of buckets (see Figure 4).

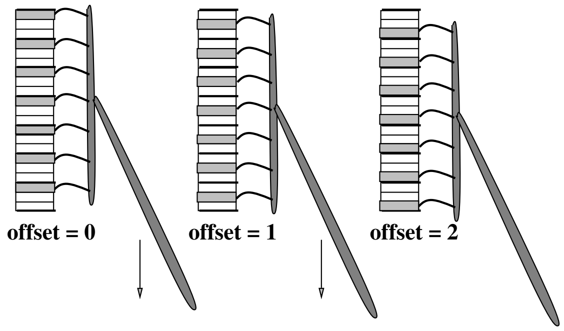


Figure 5: Loop Raking. Vectors are loaded with a stride of $s = N/L$. To process an entire array of length N, we move the rake down one step on each of s iterations.

Loop Raking: The typical way to vectorize an operation on an array uses *strip mining* [16]:

do $i \leftarrow 0$ **to** $N/L - 1$ **by** L
  process items $\langle i$ to $i + L - 1\rangle$

where L is the vector-register length. In this method an element of a vector-register handles every Lth element in the array. The problem with using strip mining for vectorizing counting sort is that the sort will not be stable, as required for radix sort to be correct. This is because each virtual processor (element of the vector register) will be responsible for a strided set of keys rather than a contiguous block of keys as required by the parallel algorithm.

To generate a stable counting sort, we introduce a technique that we call *loop raking*. Loop raking uses a constant stride (s) of N/L to access a set of elements that are evenly distributed across the input vector—as if a rake was placed over the vector and shifted on each vector load. See Figure 5. The loop structure is as follows:

$s \leftarrow N/L$
do $i \leftarrow 0$ **to** $s - 1$
  process items $\langle\, i, i+s, i+2\cdot s, \ldots, i+(L-1)\cdot s\rangle$

To avoid excessive bank conflicts during a vector load, L is selected so that s is odd (see discussion of shape factors in [4]). To extend loop raking to multiple processors, a vector is divided into P parts, and each part is raked by one of the processors. As will be discussed in Section 4, loop raking is used in all three phases of the counting sort.

Processor Memory Layout: On the CRAY Y-MP, there are B (256) separate banks of interleaved memory, each with a relatively high latency. If one access to a bank is followed directly by another access to the same bank, the second access will be delayed. A memory location X is contained in bank X mod B, causing sequential accesses to be very efficient, since locations are striped across the memory banks [17]. But when performing data dependent access to the buckets in the radix sort, memory accesses may refer to the same banks. Ideally, we would like to arrange the buckets so that each access would be guaranteed to use a different bank, making the sorting time independent of the distribution of the keys.

As discussed in Section 2, the buckets logically form a two dimensional array: the row index identifies a set of buckets for a virtual processor and the column index selects a bucket within a set. The obvious way to lay out the buckets would be in row-major order, i.e., with the buckets for each virtual processor stored in contiguous memory locations. This approach makes addressing the buckets simple, but causes data-dependent bank conflicts. For example, with a radix of $2^8 = 256$ and 256 memory banks, a virtual processor with a value of i will access memory bank i (assuming the buckets start at bank 0). If several virtual processors have the same value, they will access the same memory bank and introduce a delay. In fact, the worst case is quite common, since users will often sort keys in which all the keys have an identical digit.

Instead we lay out the buckets in column-major order so that the buckets used by each virtual processor are in a separate memory bank. For example, vector element 0 might only access banks 0 and 64, while vector element 1 only accesses banks 1 and 65. Hence using column-major order guarantees that each memory address in a vector gather or scatter is in a different bank. The only disadvantage to a column-major layout is that addressing bucket i requires multiplying i by the number of sets of buckets (i.e., the number of virtual processors). Fortunately, we can perform the address arithmetic with inexpensive bit shift instructions, since the number of sets of buckets is a power of 2. Furthermore, this shift can be folded in with other operations needed to extract the current digit from the keys.

4 CRAY Implementation of Radix Sort

This section describes our implementation of parallel RADIX-SORT on an 8-processor CRAY Y-MP. Each pass of RADIX-SORT consists of a call to EXTRACT-DIGIT followed by the three phases of the COUNTING-SORT algorithm: HISTOGRAM-KEYS, SCAN-BUCKETS and RANK-AND-PERMUTE. For each routine, this section discusses vectorization and parallelization and develops an equation for estimating the execution time. This section also discusses our implementation of a more general sorting interface.

We implemented RADIX-SORT using our own macro assembler, written in LISP. The macro assembler generates CRAY Assembly Language (CAL). In the assembler, we developed a set of macros for implementing loop raking, including a version that generates an unrolled loop to avoid register reservation effects [17] (therefore allowing better chaining), and a version that supports **doall** loops by generating calls to microtasking macros ($MDO) provided from the CRAY macro package [7]. Part of the motivation for developing the macro assembler was to allow us to easily port the code to other vector machines.

Extract Digit: On each pass, RADIX-SORT extracts the current digit from the keys. In the parallel histogram operation, each processor then uses this digit to compute an index into the array of buckets. Our implementation of EXTRACT-DIGIT folds together the two operations of extracting the digit and converting it to a bucket index. This simplifies the HISTOGRAM and RANK-AND-PERMUTE routines, by providing direct access to the buckets without requiring additional address computation.

Since each virtual processor will process a contiguous block of keys, EXTRACT-DIGIT uses loop raking. The digits are extracted by masking with a bitwise AND operation and shifting the result to the low order bits. The digit must then be converted into an index into the column-major bucket array, where bucket m of virtual processor n is stored at an offset of $m \cdot L \cdot P + n$. The column offset is computed by multiplying the digit by the number of sets of buckets, $L \cdot P$. The row offset is computed by having each virtual processor add its virtual processor number n (which is precomputed outside the loop).

The EXTRACT-DIGIT routine is quite efficient. Multiplying by $L \cdot P$ can be computed with a shift (since $L \cdot P$ is a power of two), and the mask, shift, and addition are all chained in the hardware pipeline. The time for EXTRACT-DIGIT is

$$T_{\text{Extract-Digit}} = 1.2 \cdot N/P$$

where the constant was determined empirically. The units for all equations in this paper are expressed in clock cycles (1 clock cycle = 6 nsec).

Histogram Keys: The first step to building the histogram consists of clearing the buckets using a simple **doall** loop. There are $2^r \cdot L \cdot P$ buckets, and we achieve a speedup of P, yielding the equation:

$$T_{\text{Clear-Buckets}} = 1.1 \cdot 2^r \cdot L \cdot P/P = 1.1 \cdot 2^r \cdot L$$

To produce counts of the digits, HISTOGRAM-KEYS uses loop raking to access the digits. See Figure 4. Since each digit is already converted to a bucket index, the counts are simply gathered from the buckets, incremented, and scattered back to the buckets.

In the pseudo-code below, the variables $V1$, $V2$ and $V3$ are vector registers. The notation $X[first : last : stride]$ identifies elements from $first$ to $last$, incrementing by $stride$. The access $X[i : i+(L-1)s : s]$ therefore loads the L elements $\langle i, i+N/L, i+2N/L, \ldots, i+(L-1)N/L \rangle$. To implement loop raking, each physical processor initializes a variable $offset$ to the starting location of the keys which it will process. Then the stride s is set to the number of elements per virtual processor, since each vector register element will simulate one virtual processor.

HISTOGRAM-KEYS
  % $B[0..(L \cdot P \cdot 2^r - 1)]$ = Buckets
  % $D[0..N-1]$ = Digits (pre-converted to bucket indices)
  doall $k \leftarrow 0$ **to** $P - 1$
    $offset \leftarrow k \cdot (N/P)$
    $s \leftarrow \frac{N}{L \cdot P}$ /* set stride for doing histogram */
    do $i \leftarrow offset$ **to** $offset + s - 1$
        /* load bucket index with stride s */
      $V1 \leftarrow D[i : i + (L-1)s : s]$
      $V2 \leftarrow B[V1]$ /* gather counts from buckets */
      $V3 \leftarrow V2 + 1$ /* increment counts */
      $B[V1] \leftarrow V3$ /* scatter new counts to buckets */

The execution time for HISTOGRAM-KEYS is:

$$T_{\text{Histogram-Keys}} = 2.4 \cdot N/P$$

Scan Buckets: As discussed in Section 2, SCAN-BUCKETS in the parallel algorithm is equivalent to executing a scan on the bucket matrix in column major order. In our implementation, the buckets are already arranged in column major order, so we only need to execute one scan on the entire matrix.

In a previous paper, we described in detail how to vectorize scan operations [4]. The algorithm is reviewed here. A parallel algorithm for scanning a vector of length *N* on *P* processors has three steps:

1. Processor i sums a contiguous block of elements, is through $(i+1)s - 1$, where $s = N/P$. Call this result Procsum[i].

2. The vector Procsum is scanned using a tree-summing [12] or similar algorithm.

3. Processor i sums its portion of the vector, starting with Procsum[i] as the initial value, and accumulating partial sums. The partial sums are written out to the corresponding locations.

To vectorize the parallel algorithm, consider each element of a vector register to be a virtual processor. Then, in the above scheme, element *i* of the vector register would need to sum elements is through $(i+1)s - 1$ of the vector being scanned. This operation can be vectorized by loop raking, i.e., traversing the vector with stride s, and using elementwise vector instructions to sum the sections. In Step 2, the scan across virtual processors is computed with a scalar loop, since executing one pass over the Procsum vector accounts for only a small fixed cost.

The vectorized scan operation can be parallelized with a few modifications. The parallel scan algorithm we developed is equivalent to simulating the vector algorithm with a vector length of $L \cdot P$. In the three step scan algorithm, step 1 is the same for the parallel algorithm, since the operations for each virtual processor are independent. Step 2 begins by having each processor scan the sums computed by step 1, as with the single-processor algorithm. Each processor then writes out the sum of its elements, and one processor performs a serial scan on the P processor sums. The result of this scan is added to the sums computed in step 1. The rest of the scan proceeds as with the single-processor algorithm.

The execution time for SCAN-BUCKETS is:

$$T_{\text{Scan-Buckets}} = 2.5 \cdot 2^r \cdot L \cdot P/P = 2.5 \cdot 2^r \cdot L$$

Permute Keys: This phase is very similar to HISTOGRAM-KEYS, except that the keys are also loaded and permuted (scattered) using the offsets gathered from the buckets.

Unlike the accesses to buckets, permuting the keys can cause bank conflicts. On the CRAY Y-MP, the time to permute a vector with the scatter instruction depends on the permutation, since some permutations cause more bank conflicts than others. The best case, an identity permutation that causes no bank conflicts, takes about 1.3 cycles/element. The worst case takes about 5.5 cycles/element, and a random permutation takes approximately 1.8 cycles/element. For the analysis, we base our timing estimates on sorts of random keys. These estimates are accurate in practice because loop raking tends to break up common patterns that would cause the worst case permutations, and our implementation performs well on random inputs and regular inputs, such as a pre-sorted array. The time for the RANK-AND-PERMUTE step is approximately:

$$T_{\text{Rank-And-Permute}} = 3.5 \cdot N/P$$

Orders: In practice, a sorting routine that can carry data along with keys or sort records is more useful than one that just sorts keys. A clean way to provide a more general sorting interface is to implement a routine that returns a *permutation* that can be used to sort data. We implemented an ORDERS routine that returns a permutation $Perm$, such that $Perm[i]$ is the index of the ith smallest key. An ORDERS routine is useful for implementing a multipass sorting routine for sorting records, since the permutations returned from sorting on each field of the record may be composed. ORDERS also makes sorting large data items with small keys more efficient, since only the keys are manipulated internally. An ORDERS routine is, in fact, the standard library routine callable from Fortran on the CRAY [8].

To transform our sorting implementation into an ORDERS routine, we store a *current permutation* and permute it on each pass rather than the keys. We start by initializing the current permutation, $Perm$, to the identity permutation, such that $Perm[i] = i$. We modify the HISTOGRAM routine to gather the source data using the current permutation, rather than loading it directly, and we modify the RANK-AND-PERMUTE routine to permute the $Perm$ array rather than the data itself. Because the source keys are gathered rather than loaded, $T_{\text{Extract-Digit}}$ for ORDERS is approximately 2 cycles per key, as compared to 1.2 cycles for SORT.

5 Performance Analysis

This section presents an analysis of our sorting implementation. We first develop equations that predict the running time in terms of various parameters. These equations allow us to calculate the optimal values of free parameters, such as the radix and vector length, and allow us to predict the running time on any number of processors. The equations also allow us to factor out parameters specific to our implementation and provide a degree of machine independence to our analysis. This section also describes the space used by the implementation and presents performance measurements, including a comparison of our ORDERS routine to the CRAY library implementation and speedup measurements on multiple processors of the CRAY Y-MP.

Equations

The sorting time can be characterized by two main parameters of the implementation, the time per bucket, T_{bucket}, and the time per key, T_{key} (expressed in clocks per element per processor):

$$\begin{aligned} T_{\text{bucket}} &= T_{\text{Clear-Buckets}} + T_{\text{Scan-Buckets}} \\ &= 3.6 \\ T_{\text{key}} &= T_{\text{Extract-Digit}} + T_{\text{Histogram-Keys}} + T_{\text{Rank-And-Permute}} \\ &= 7.1 \end{aligned}$$

where the constants are taken from the previous section. Since there are $L \cdot 2^r$ buckets per processor and N/P keys per processor, the time for one call to $T_{\text{Counting-Sort}}$ (including the time for EXTRACT-DIGIT) is:

$$T_{\text{Counting-Sort}} = L \cdot 2^r \cdot T_{\text{bucket}} + N/P \cdot T_{\text{key}} \quad (2)$$

Sorting b-bit keys requires b/r calls to COUNTING-SORT:

$$T_{\text{Radix-Sort}} = \frac{b}{r}(L \cdot 2^r \cdot T_{\text{bucket}} + N/P \cdot T_{\text{key}}) \quad (3)$$

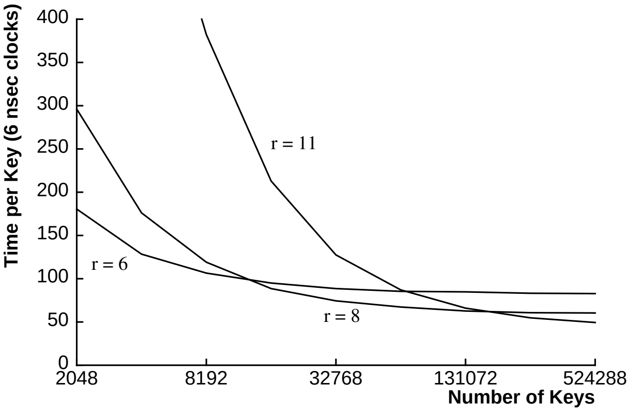


Figure 6: Time per key for a 64-bit sort on one processor as the number of bits per pass (r) is varied. As the number of keys is increased, the optimal value for r increases.

Choosing the Radix: The parameter r should be chosen to minimize the total sorting time. As Figure 6 indicates, the optimal value for r increases with the number of elements per processor. For a given problem size, choosing r below the optimal value will cause too much work on keys, while choosing r above the optimal value will cause too much work on buckets, as illustrated in Figure 7. We can determine the value for r that minimizes the total time by differentiating formula 3 with respect to r and setting it equal to 0. The value for r that we obtain satisfies:

$$r = \lg(N/P \cdot \frac{T_{\text{key}}}{L \cdot T_{\text{bucket}}}) - \lg(r \ln 2 - 1) \quad (4)$$

$$\approx \lg(N/P) - \lg(L) - 1 \quad (5)$$

where we use the fact that $T_{\text{key}} \approx 2T_{\text{bucket}}$ and we approximate the second term of equation 4 with the constant 2, derived from setting r to a moderate value of 7. (Throughout the paper, $\lg n$ denotes $\log_2 n$.)

For $N = 32K$, as in Figure 7, the optimal value for r predicted by equation 5 is $r \approx 8$ which, in fact, minimizes the total time.

When we substitute the approximation for r back into equation 3, we obtain the following approximation for the total sort time:

$$T_{\text{Radix-Sort}} \approx \frac{b \cdot (N/P)}{\lg(N/P) - \lg(L) - 1} \left(\frac{T_{\text{bucket}}}{2} + T_{\text{key}} \right) (6)$$

Figure 8 compares this equation with the actual running time of our implementation of RADIX-SORT in which the best experimental values for r are used. As the figure indicates, this equation accurately predicts the running time. Notice that the time per key *decreases* as we increase N,

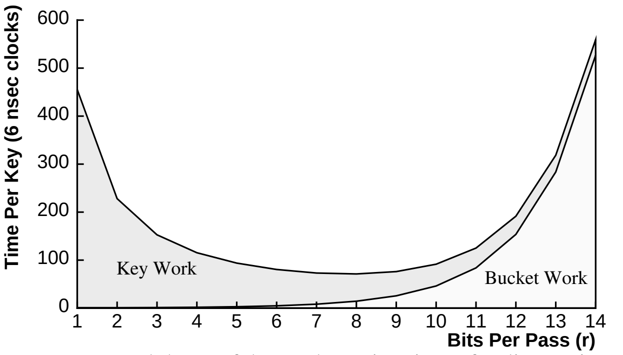


Figure 7: Breakdown of the total running time of radix sort into the time spent on buckets and time spent on keys. Times are derived from two terms of equation 6 for a one-processor 64-bit sort on 32K elements. As r is increased, the work per bucket increases and the work per element decreases. For the parameters chosen, the optimal value for r is 8, as indicated by the top curve representing the total time.

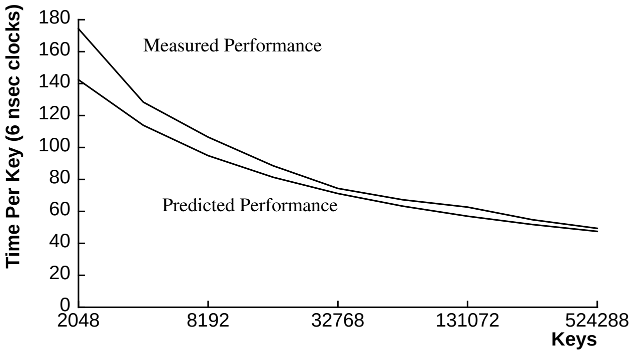


Figure 8: Predicted and measured performance of radix sorting 64-bit keys on one processor. Measured performance uses the empirically determined optimal values for r. The predicted performance was calculated using formula 6.

the number of keys, because we can increase the radix. In contrast, the time per key for comparison-based sorts is typically proportional to $\lg N$, and thus increases for larger sorting problems. Furthermore, the time for radix sort is proportional to the number of passes of COUNTING-SORT, or the number of digits. Thus, we can sort 32-bit keys approximately twice as fast as 64-bit keys, and 20-bit keys approximately three times as fast. In contrast, comparison-based sorts typically take the same amount of time for shorter keys. Shorter keys are important because users will often sort indices or pointers that are significantly shorter than 64 bits.

Vector Length: Another approach to optimizing the balance between computation on buckets and computation on keys would be to vary the vector-register length, i.e., reduce the number of virtual processors. This would cut down the time for clearing and scanning the buckets, T_{bucket}, at the expense of increasing the cost per element for performing the histogram, T_{key}. We determined experimentally that cutting the vector length L in half to a length of 32 causes a 20% increase in T_{key}.

At high values of N/P, and thus high values of r, increasing r by a constant amount has little effect on the number of calls to COUNTING-SORT. However, by using equation 6 to approximate the total sorting time, we can determine when reducing the vector length to 32 would save enough passes of radix sort to balance the increased time per key. We substitute $L = 32$ for the vector length and the corresponding higher value for T_{key} into equation 6. Comparing the resulting expression to the total time for a vector length of 64 yields the inequality $N/P < 9000$. We conclude that reducing the vector length only yields improvements for small sorting problems ($N/P < 9000$), as we have confirmed experimentally.

Speedup on Multiple Processors: Equations 5 and 6 can be used to predict the theoretical speedup on P processors. Notice in equation 6 that the total time is a function of the number of elements per processor; the radix is chosen based on N/P and the total time is proportional to N/P. This implies that we will get linear speedup when increasing the number of processors if N/P is held constant, i.e., if the problem size grows with the number of processors. However, if we fix the problem size, N, as we increase the number of processors, we do not quite get a speedup of P, because the optimal radix is lower. For example, with $N = 256K$, the optimal value for r on 1 processor is $\lg(N/1) - 7 = 11$, while the optimal value on 8 processors is $\lg(N/8) - 7 = 8$. Thus with 1 processor, we need 6 calls to COUNTING-SORT to sort 64-bit keys, but with 8 processors, we need 8 calls, yielding a speedup of 6 on 8 processors.

Space: An important concern for designing a practical sorting routine is to minimize the space required by the implementation as well as the running time. To sort an array of size N, our sort requires a temporary array of size N for the result of extracting the current digit, an array of size N for the destination of the permute, and an array of size $L \cdot P \cdot 2^r$ for the buckets. Since the optimal value of r is approximately $lg(N/P) - 7$, the amount of memory for the buckets will be approximately $L \cdot P \cdot 2^{lg(N/P)-7}$, which simplifies to $\frac{N}{2}$. Thus the total amount of memory used besides the source is approximately $2.5N$.

There are several ways that memory could be saved at the expense of increased execution time. Instead of extracting the current digit and placing it in a temporary array, we could extract it as needed in the HISTOGRAM and RANK-AND-PERMUTE steps. We could also save memory on the buckets by using a lower radix or reducing the vector length.

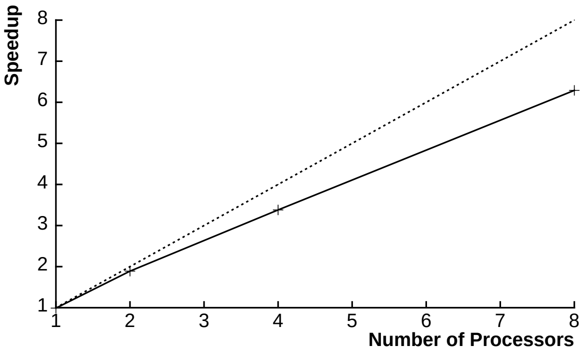


Figure 9: Speedup on multiple processors for a 64-bit sort. The number of elements per processor is fixed at 64K as the number of processors is varied.

Measurements

Timing measurements were taken with the `rtclock()` system call, and multiple measurements were taken to smooth out the effects of the presence of other users on the system. All timings were taken on random 64-bit keys.

We begin by comparing our implementation of ORDERS to the CRAY library routine, which is also a radix sort, in order to show that the results we obtain are due to an algorithmic improvement, rather than tighter code. According to the CRAY manual page [8], "ORDERS has been optimized in CAL to make efficient use of the vector registers and functional units at each step of a pass through the data." The library routine uses either $r = 8$ or $r = 16$ as determined by the caller, and judging from the temporary memory requirements, it does not use multiple sets of buckets. Figure 1 graphs the performance of our implementation and the library implementation. As the figure indicates, our implementation is 3 to 5 times faster for sorting more than 2K items. The library routine is more space efficient, however.

Figure 9 shows the speedup we obtained on multiple processors of the CRAY Y-MP, holding the number of elements per processor constant as the number of processors is varied. Using 8 processors, we achieve a speedup of approximately 6.3. We do not get an optimal speedup of 8 primarily because of memory contention. We can approximate this effect by dividing the total sorting time by a memory contention term

$$(1 - .22(P/P_m)^{1.5})$$

where P is the number of physical processors used and P_m is the maximum number of physical processors (8 on the

CRAY Y-MP). We derived this term purely empirically by fitting a curve to the data. Using this approximation, the total sorting time in 6 nsec clock cycles (for a vector length of $L = 64$) is:

$$T_{\text{Radix-Sort}} = \frac{b}{r}(230 \cdot 2^r + 7.1 \cdot \frac{N}{P})/(1 - .22(P/P_m)^{1.5})$$

Thus, fully linear speedup is not achieved due to two effects: saturating the memory system causes a small degradation in memory bandwidth, and using a separate set of buckets for each virtual processor forces us to use a slightly lower radix.

6 Conclusions

We designed a radix sort algorithm for vector multiprocessors and demonstrated that radix sort is extremely efficient in practice for the CRAY Y-MP. We found that starting with a parallel algorithm and converting it to a vector multiprocessor algorithm yielded several benefits. The parallel radix sort algorithm exposed a large degree of parallelism. By treating each vector element as a virtual processor, we exploited two levels of parallelism, using a unified approach to vectorization and parallelization. Furthermore, the clear separation of local and global computation in the parallel algorithm allowed us to optimize memory accesses on the CRAY. When accessing buckets, we used the concept of local memory from the parallel algorithm to assign the buckets for each virtual processor to separate memory banks. In our loops, we used *loop raking* to simulate the access pattern of the algorithm by assigning a contiguous section of data to each vector element. Finally, we found that scan primitives that are often used on parallel computers such as the CM-2, provide a powerful tool for algorithm design and implementation [2]. We extended our previous work on vectorizing scans [4] and demonstrated that they can be implemented efficiently on vector multiprocessors.

Acknowledgements

We thank the Pittsburgh Supercomputing Center for use of their CRAY Y-MP, Sid Chatterjee, Margaret Reid-Miller, Jay Sipelstein, Kurt Thearling, and Leo Unger for their helpful comments on this work, and Qi Lu for his work implementing the scan and other vector operations on the Ardent Titan.

References

[1] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[2] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11):1526–1538, November 1989.

[3] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Gregory Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 3–16, Hilton Head, SC, July 1991.

[4] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Proceedings Supercomputing '90*, pages 666–675, November 1990.

[5] Richard Cole and Uzi Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *Proceedings ACM Symposium on Theory of Computing*, pages 206–219, 1986.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 1990.

[7] Cray Research Inc., Mendota Heights, Minnesota. *Macros and Opdefs Reference Manual, SR-0012D*, June 1988.

[8] Cray Research Inc., Mendota Heights, Minnesota. *ORDERS(3SCI) Manual Page SR-2081 5.1*, March 1988.

[9] Cray Research Inc., Mendota Heights, Minnesota. *Symbolic Machine Instructions Reference Manual, SR-0085B*, March 1988.

[10] C. A. R. Hoare. Quicksort. *Computer J.*, 5(1):10–15, 1962.

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

[12] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the Association for Computing Machinery*, 27(4):831–838, October 1980.

[13] Stewart A. Levin. A fully vectorized quicksort. *Parallel Computing*, 16:369–373, 1990.

[14] B.M.E. Moret and H.D. Shapiro. *Algorithms from P to NP*. Benjamin Cummings Publishing Company, 1990.

[15] Lasse Natvig. Logarithmic time cost optimal parallel sorting is *not yet* fast in practice! In *Proceedings Supercomputing '90*, pages 486–494, November 1990.

[16] D. A. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[17] Kay A. Robbins and Steven Robbins. *The Cray X-MP/Model 24: A Case Study in Pipelined Architecture and Vector Processing*. Springer-Verlag, 1989.

[18] Wolfgang Rönsch and Henry Strauss. Timing results of some internal sorting algorithms on vector computers. *Parallel Computing*, 4, 1987.