Single Buffered Histogram Sort

by Marion McCoskey

Thursday, February 04, 1999

I have discovered a general-purpose, single buffered sorting algorithm that is faster than Quick Sort and insensitive to the distribution and ordering of the data to be sorted.

The Histogram Sort has other advantages in addition to being faster and more stable:

1. A part of the data is completely sorted before the entire sort is complete. There are at least two ways this property of the Histogram Sort can be used to advantage.
   a) One processor can pass completely sorted data to another processor while the rest of the data set remains partially sorted.
   b) A percentile-based subset of the data can be sorted, and the rest left partially sorted. This can result in a speed increase of several hundred percent over a complete sort. Queries such as: "Who are the top 10 salespeople of XYZ Corporation?" or "List 20 people who make median salaries at XYZ Corporation." Don't require that the entire record set be sorted with the Histogram Sort.
2. The integrated histograms generated during the sort process can also be used as hash tables to speed up the search part of a search and sort procedure such as that required by compilers, assemblers, interpreters, and data base joins.
3. Unlike the Quick Sort the Histogram Sort does not suffer performance degradation on some orderings of data. On systems without caching, the Histogram Sort should take the same time, no matter what the original order. On systems with caching, the Histogram Sort is slightly faster on data sets that are more nearly sorted.
4. The histogram algorithm is amenable to hardware acceleration. The Quick Sort has been known for decades. Existing hardware has been designed with the idea that a processor that runs Quick Sort faster has a competitive advantage. The Histogram Sort allows re-thinking of some of our basic assumptions about the design of computer hardware

Lines	Bubble	Shell	Quick	Histogram
2900	40.7	3.3	0.0	0.0
5899	186.8	11.0	0.6	0.5
11799	790.4	29.7	1.1	0.0
23375		86.8	2.2	0.5
47036		390.0	4.9	1.7
93726		1357.7	12.1	3.9
187314			26.9	9.4
375290			59.4	20.4
749546			134.6	44.5
1499580			296.6	94.0
2999883			637.6	190.1

Figure One

Figure One shows benchmarks for the bubble sort, shell sort, Quick Sort, and single buffered Histogram Sort on randomly-ordered, zero-terminated strings. The strings are limited to the 26 English letters and the 10 numerals and the sorts are all case-insensitive. This requires only a histogram of 37 entries. One is required for the zero-termination of the strings.

The data in Figure One shows the software implementation of the Histogram Sort in a slightly better light than strings with more kinds of characters would, but many string sorts are carried on with reduced character sets, in essence wasting processor capacity for characters that are not used. In a hardware accelerated version of the Histogram Sort, histogram size would matter far less, if at all.

Lines	Qsort	qSort Worst Case	Bubble Sort	qSort Worst/Best Ratio
1000	0.0	1.6	3.3	
2000	0.0	6.6	15.9	
4000	0.6	30.7	77.4	51.2
8000	0.6	121.9	345.5	203.2
16000	1.6	426.8	1440.1	266.8
32000	3.3	1405.0	5922.6	425.8
64000	7.2	5159.1		716.5

Figure Two

Even though it was the world's fastest single-buffered sort on most data, the Quick Sort has a major drawback which is illustrated in the Figure Two. In a naïve implementation of the Quick Sort, worst case performance occurs on already sorted data. Practical Quick Sorts work around this problem by putting in a couple of lines which make the worst case performance occur on a data ordering that is unlikely to be encountered, but any probability greater than zero is troublesome in many situations making the Quick Sort unsuitable for some applications, despite its speed.

One day while writing a program for my hobby of studying esoteric musical scales, I needed a sort. Following my usual practice, I wrote a shell sort, but it was slow enough that it gave a noticeable delay to the program I was writing. To speed up that sort, I used a hashing method only applicable to that particular situation.

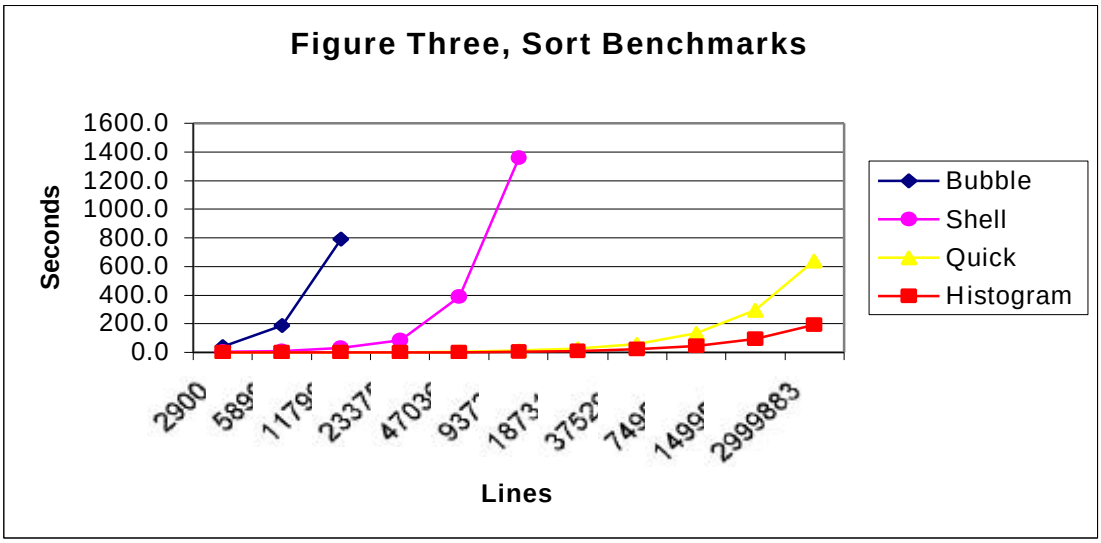
It occurred to me that it would be wonderful if there was a general purpose hash function that could speed up all sorts, and that an integrated histogram was just such a function.

So I filed a patent on a sort method I called the Histogram Sort. I recently discovered that the basic principle of the Histogram Sort is already well known as the count sort, but the way I applied that principle is apparently novel, and better in many ways than the way the count sort principle is traditionally applied.

The basic count sort is very efficient, but for practical problems it would require prohibitively large histograms. The conventional method of decomposing the algorithm is the Radix Sort. The Histogram Sort differs from the Radix Sort in the method used to apply the basic count sort algorithm to large problems.

The method I described in my patent was a double-buffered method, but I have recently managed to make a slight variation on the algorithm and execute the sort in a single buffer. For intellectual property reasons, I am unable to discuss details of the single buffered version of my algorithm here, but I will present some benchmarks. I can also say that the single-buffered version requires a data structure other than the nested histograms, but this data structure is small and it is not proportional to the size of the data to be sorted. It turns out that the single buffered method is actually a little faster than the double buffered method. I have coded both variable-length string, and fixed - length numerical versions of the single buffered algorithm. I am including a source listing for a double-buffered numerical Histogram Sort with this package.

The Histogram Sort makes first pass counting data items that fall into a primary region determined by the most significant bits of their keys. Then the data items are distributed according to these key bits, and the histogram is saved. In a string implementation, all the strings that start with "a" are put in one primary region, all the strings that start with "b" are put in another primary region, etc. The first two elements of the integrated histogram define the start and end of the first region which would contain all the strings starting with "a", for example.



Next this first “a” region is divided into a number of smaller secondary regions by a process like that applied to the entire data set, except the subsequently significant key bits are used to create the second nested histogram. All the strings starting with “aa” are put in one region, and the strings starting with “ab” are put in another region. This process is repeated using less significant bits for each iteration until the entire first primary region is sorted. Then the process is applied to subsequent primary regions until all the data is sorted.

The Histogram Sort is easily adapted to sorting arrays of pointers to zero terminated strings. Such an adaptation for the straight Radix Sort would be very difficult.

My patent application describes a method of hardware acceleration for the Histogram Sort that should make it by far the fastest sorting algorithm on the planet.

Without hardware acceleration, the Histogram Sort algorithm becomes less efficient than other algorithms for sorting small data sets which produce extremely sparse histograms. This is because it must clear each histogram before it is used, and then process all the entries in each histogram. Thus it may be clearing histogram elements that are not useful in the sorting process, and then checking all these elements to see if they contain data.

A hardware accelerator would use special histogram memory that could be cleared in a single clock pulse. This memory would have all the clear inputs of the memory registers connected to a single signal, thus nearly eliminating the time it takes to clear a histogram.

The hardware accelerator would also use a priority encoder to go directly to a histogram entry containing data, taking just a few gate delays to skip over any number of histogram entries that do not require processing.

There are other techniques that are already well know such as zero overhead loops and data pre-fetch that could also be applied to reduce the Histogram Sort run time in a sort coprocessor.

When implementing the Histogram Sort in software the sparse histogram problem can be ameliorated by simply sorting small partitions with some other method.

You can get more information about the Histogram Sort by sending e-mail to mckyyy@aol.com.

Comparison between the Straight Radix Sort and the Histogram Sort		
The Straight Radix Sort is described in the book "Algorithms" by Robert Sedgewick		
Radix Sort	Histogram Sort	Comment
Double buffered only	Single or double buffered	Double buffering requires twice as much RAM. If the data set is large enough to require disk access, this can be a major performance problem.
Does not partition data	Partitions data	All bits of all data items have to be processed if there is no partitioning. With partitioning different sorting strategies can be applied to different partitions according to their size, which is known in advance.
Uses only one histogram	Uses nested histograms	Nested histograms require more storage, but the requirement is very small compared to the size of the data to be sorted.
Does most significant bits first	Does least significant bits first	Doing the least significant bits first means that the data remains completely unsorted until the last pass. Doing the most significant bits first means that it is only necessary to process enough data bits in any particular item to distinguish it from all the other items in the data set, and completely sorted data becomes available before the entire sort is complete.
Extremely difficult to apply to variable length data.	Easily applied to variable length data.	Starting with the least significant bits makes applying the straight Radix Sort to variable length strings extremely difficult.
Figure Four		

In his book "Algorithms", Robert Sedgewick presents two Radix Sorting methods; a single buffered method he calls the "radix exchange sort", and a double-buffered method he calls the "straight Radix Sort".

The single-buffered radix exchange sort Sedgewick describes bears little relationship to the Histogram Sort described in my patent.

As seen in Figure Five, the speed advantage of the numerical Histogram Sort is not as great as with the string sort.

Lines	Two Buffers	One Buffer	Quick Sort
	Histogram Sort		
125000	1.7	1.1	1.6
250000	2.8	1.6	3.3
500000	4.9	4.4	6.6
1000000	11.6	9.4	14.3
2000000	24.2	21.5	30.7
4000000	48.9	45.6	67.5
8000000	96.1	78.0	140.6
Figure Five			

Sorting is very important in computer science. It has been estimated that as much as 25% of all computer time is spent in sorting, with the proportion going up for larger and more expensive machines doing database work and compiling large programs.

A general purpose sorting algorithm that is fast and insensitive to the distribution and ordering of the data to be sorted. Is an important development.

Serious consideration should be given to embedding the Histogram Sort algorithm into a hardware accelerator and making it an industry standard. Not only would machine time be saved in the sort process, but programmer/analyst time would be saved in the code generation process.