RC 20446 (05/01/96)
Computer Science/Mathematics

# IBM Research Report

## A Super Scalar Sort Algorithm for RISC Processors

Ramesh C. Agarwal

IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598



IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

# A Super Scalar Sort Algorithm for RISC Processors

**Ramesh C. Agarwal**
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
agarwal@watson.ibm.com

## Abstract

The compare and branch sequences required in a traditional sort algorithm can not efficiently exploit multiple execution units present in currently available high performance RISC processors. This is because of the long latency of the compare instructions and the sequential algorithm used in sorting. With the increased level of integration on a chip, this trend is expected to continue. We have developed new sort algorithms which eliminate almost all the compares, provide functional parallelism which can be exploited by multiple execution units, significantly reduce the number of passes through keys, and improve data locality. These new algorithms outperform traditional sort algorithms by a large factor.

For the Datamation disk to disk sort benchmark (one million 100-byte records), at SIGMOD'94, Chris Nyberg et al presented several new performance records using DEC alpha processor based systems.

We have implemented the Datamation sort benchmark using our new sort algorithm on a desktop IBM RS/6000 model 39H (66.6 MHz) with 8 IBM SSA 7133 disk drives (total cost $73K). The total elapsed time for the 100 MB sort was 5.1 seconds (vs the old uni-processor record of 9.1 seconds). We have also established a new price performance record (0.2¢ vs the old record of 0.9¢, as the cost of the sort). The entire sort processing was overlapped with I/O. During the read phase, we achieved a sustained BW of 47 MB/sec and during the write phase, we achieved a sustained BW of 39 MB/sec. Key extraction and sorting of one million 10-byte keys took only 0.6 second of CPU time. The rest of the CPU time was used in moving records, servicing I/O, and other overheads.

Algorithmic details leading to this level of performance are described in this paper. A detailed analysis of the CPU time spent during various phases of the sort algorithm and I/O is also provided.

## 1 Introduction

In 1985 [1], a group of database experts defined three basic benchmarks to measure the transaction processing performance of computer systems. One of these benchmarks does a disk to disk sort of one million records of 100 bytes each. The relevant measures are elapsed time and cost. The following is a direct quote from [1].

"The sort benchmark measures the performance possible with the best programmers using all the mean tricks in the system. It is an excellent test of the input/output architecture of a computer system and its operating system.

The definition of the sort benchmark is simple. The input is 1 million records stored in a sequential disk file. The first 10 bytes of each record are the key. The keys of the input file are in random order. The sort program produces an output file containing the input sorted in key order. The sort may use as many scratch disks and as much memory as it likes."

The elapsed time is the time from start to the end of the sort program and cost is the time-weighted cost of the hardware and software packages used in the sort. The article uses a five year cost averaging and a second costs about 6.3E-9 of the five year capital cost. Thus if a workstation costing $100K takes 10 seconds to do the benchmark, then cost of 100 MB sort is 0.63¢.

## 2 Prior work on the sort benchmark

Over last ten years, several authors [2]-[10] have presented their results on this sort benchmark. Nyberg et al. [9]-[10] have given an excellent summary of prior work. During these ten years, the time required to do this benchmark has reduced from an hour to just a few seconds, an improvement by three orders of magnitude or roughly by a factor of two every year. The cost of sort has also reduced from $4.61 to under a penny which is roughly a 2x improvement every year.

At SIGMOD'94, Nyberg et al. [10] presented several results which established new records in many categories. The best single processor performance of

9.1 seconds was obtained on a 200 MHz DEC-7000-AXP system using 16 disk drives. The system cost was $247K resulting in a cost of 1.4¢ for the 100 MB sort. The best price/performance record of 0.9¢ was obtained on a 150 MHz DEC-3000-AXP using 10 disk drives. The system cost was $97K and the benchmark took 13.7 seconds. Our results significantly improve on both these numbers using a 66.6 MHz RS/6000 desktop workstation model 39H with 256 MB of memory, 128 KB of L1 cache, no L2 cache, and 8 IBM 7133 1.1 GB SSA disk drives. The sort application needed only 160 MB of memory to run. Therefore, for the same system cost, we could sort a bigger file (approximately 180 MB or so). The base cost of the system with 64 MB of memory is $31K. The overall system cost including all the hardware and software is $73K. On this system, the benchmark ran in 5.1 seconds, resulting in the sort cost of about 0.2¢. This is more than a factor of four cheaper compared to the previous best result.

In the same paper, they also presented results on shared memory systems. They created a new record time of 7.0 seconds on a 200 MHz, 3-CPU DEC-7000-AXP system using 28 disk drives. The cost of this system was $312K. Our results have improved on this time by a factor of 1.37 on a system costing five times less. However, our results are not the fastest ever reported. That distinction goes to SGI. At SIGMOD'95 they [11] reported a time of 3.5 seconds on a 12-CPU Challenge XL system using 96 disk drives. SGI did not report the system cost. Chris Nyberg of Ordinal Technology provided the sort software. At that time, SGI also announced a new record of 1.6 GB for the minute sort benchmark [9]-[10] on the same system. The estimated system cost of this machine is $700K (could be off by 2x either way). This was the system cost to do the 1.6 GB sort. Clearly, to do just the 100 MB Datamation sort benchmark will require less memory and therefore a somewhat cheaper system. For the purpose of this paper, we will ignore the system cost for the SGI machine. The table below summarizes the results reported at SIGMOD'94, SIGMOD'95, and our results.

## 3 Overview

Because memory is cheap, it is reasonable to implement a one-pass sorting for a 100 MB file. All recent implementations of the benchmark have used a one-pass approach to sorting. This requires a main memory of slightly over 100 MB. Our implementation required 116 MB to run the benchmark and some additional memory for the operating system for a total of 160 MB. There are two distinct phases of the sort benchmark. During the first phase (also called the read phase), the 100 MB data file is read. Clearly, we can not begin to sta writing the output file (second phase - writing phase) until the input file is fully read. Thus, there is no opportunity to simultaneously do disk reads and disk writes.

The minimum sort time is the time required to read a 100 MB file from disk and to write the sorted file back to disk. Our goal was to come very close to this minimum. This requires overlapping almost all the sort processing with I/O. In addition, we should use the highest bandwidth I/O supported by the system. We deliberately chose a low cost desktop system (RS/6000 model 39H) to keep the system cost low. We also decided to use as few disks as possible, without compromising on the I/O bandwidth. We chose IBM's 7133 SSA disk storage sub-system model 500 (a stand alone tower) with eight 1.1 GB disk drives. Each of these disks is capable of sustaining an I/O rate of about 7 MB/sec. Four of these disks were connected to an IBM SSA 4-port adapter which attaches to the RS/6000 microchannel. Two such adapters (total of 8 disks) were connected to the microchannel (80 MB/sec. peak). The input and output files were striped across 8 disks. The striping was handled in software. The stripe size used was a multiple of 128 KB. Steve Watts of IBM Santa Teresa Lab provided I/O kernels to do raw asynchronous I/O. With this configuration, a 100 MB disk read lasted for 2.1 seconds resulting in a sustained read bandwidth of 47 MB/sec. The write phase lasted for 2.55 seconds resulting in a sustained bandwidth of 39 MB/sec. The disk sub-system can provide a slightly higher BW during the write phase. However, during this phase, because of sort processing, the CPU was saturated.

The disk subsystem was inactive for about 0.45 second. This includes the initial launch of the sort program, opening the disk files, a gap of approximately 0.1 second between the end of the read phase and the beginning of the write phase (this is the sort processing needed before the first write block can be written out), and the final shut down of the sort program. Thus total sort time is sum of the read phase (2.1 seconds), the write phase (2.55 seconds), and an overhead of 0.45 second, for a total of 5.1 seconds. Clearly, performance of the disk sub-system was crucial to the overall performance of our disk to disk sort.

Our Datamation sort program is primarily written in Fortran with I/O kernel extensions (written in C) provided by Steve Watts of IBM Santa Teresa Lab. In all three executables are created. The first executable generates 100 byte records with 10-byte random keys and writes it out on a set of $nr$ logical volumes using a disk stripe of size $kr$*128KB where $kr$ is a an integer. The parameters $nr$, $kr$, and the number of records to be generated are chosen at the run time. The second program reads the data generated by the record generation program, using the parameters $nr$ and $kr$ (chosen during record generation phase). It sorts the file in memory and writes out the sorted file on $nw$

Table 1: Performance and price/performance of 100MB Datamation Sort Benchmark

| System | # cpu & clock MHz | controllers | drives | memory MB | time seconds | total price | disk + ctrl | ¢/sort |
|---|---|---|---|---|---|---|---|---|
| Results from [10] presented at SIGMOD'94: | | | | | | | | |
| DEC-7000-AXP | 3-200 | 7 fast-SCSI | 28 RZ26 | 256 | 7.0 | 312K$ | 123K$ | 1.4¢ |
| DEC-7000-AXP | 1-200 | 6 fast-SCSI | 16 RZ74 | 256 | 9.1 | 247K$ | 65K$ | 1.4¢ |
| DEC-3000-AXP | 1-150 | 5 SCSI | 10 RZ26 | 256 | 13.7 | 97K$ | 48K$ | 0.9¢ |
| Results from [11] presented at SIGMOD'95: | | | | | | | | |
| SGI Challenge XL | 12-CPUs | ? | 96 | 2GB | 3.5 | ? | ? | ? |
| Our Results: | | | | | | | | |
| IBM RS/6000 39H | 1-66.6 | 2 SSA | 8 7133 | 256 | 5.1 | 73K$ | 20K$ | 0.2¢ |

logical volumes using a disk stripe of size $kw$*128KB. The parameters $nw$, $kw$, and the number of records to be sorted are chosen at the run time. Finally, the third program reads the sorted file generated by the sort program and checks it to make sure that records are indeed correctly sorted. In this paper, we will describe details of the sort program. In our experiments, we obtained best performance using eight disks for reads and writes ($nr = nw = 8$), read stripe size of 256 KB, and write stripe size of 512 KB.

## 4 Read Phase

The sort benchmark is launched from AIX command line. A shared memory segment is obtained for read and write buffers. This is followed by opening of the striped input file ($nr$ logical volumes). At this point, we are roughly 0.1 second into the sort program, Now, we are ready to begin reading the input file. All read requests are initiated using a read block size of 256 KB. These are issued asynchronously and the control returns to the program. However, as part of the read request, the corresponding 256 KB block of memory is initialized by the operating system. This is also referred to as "pinning the memory" by some authors. This takes approximately 2 msec. of CPU time for a 256 KB block. We used double buffering for read requests and therefore initially two read requests were issued for each disk.

After all initial reads have been initiated, a checkio routine is called to see if any of the reads have completed. When a read is completed, another read request is immediately initiated for the corresponding device so that at all times there are two read requests for each of the logical volumes. After initiating the read request, bucket sorting (explained in a later section) is done on the block which was just read. During bucket sorting, it was decided to use 128 buckets, primarily based on the TLB consideration [15]. When this bucket sorting is completed, checkio routine is called to see if another block has arrived and the process described above is repeated.

The scheme just described extracts maximum bandwidth from the disk system by keeping them fully busy reading a large sequential file. The disk system bandwidth is primarily limited by the microchannel. However, the CPU is also fully utilized in servicing I/O, generating and initializing 100 MB addressing space for the read buffer, and in bucket sorting of 100 MB of input data. The disk read phase for the input file lasts for about 2.1 seconds resulting in a sustained disk system bandwidth of 47 MB/sec. During this period, estimated breakdown of the CPU time is as follows: initiating, servicing, and checking on I/O accounts for about 1.1 seconds, initializing 100 MB of addressing space takes about 0.75 second, and bucket sorting on 100 MB of input buffer takes about 0.25 second.

At the end of the read phase, we close all input files and now we are ready to begin the write phase. At this point, we are approximately 2.2 seconds into the sort program.

## 5 Write Phase

During the read phase, we utilized a read buffer of size 100 MB and therefore any read block of size 256 KB could be read independent of other blocks. However, during write phase, to save memory, we utilized a much smaller write buffer. We created two write buffers of size 4 MB each. Each of these buffers consisted of eight blocks of size 512 KB each; one for each disk. We overlapped writing of one output buffer with creation of the other output buffer.

In the write phase, first we open all the output files. Then we begin with completely sorting (on 10-byte keys) individual buckets (starting with bucket 0) and moving records from the input buffer to the output buffer (in the sorted sequence). This continues till a complete output buffer is obtained.

At this point, we are about 0.1 second into the write phase and now we are ready to start writing the output file. We initiate eight writes (one for each disk) for the first buffer and then resume sorting to fill the second output buffer. Now, we must wait for completion of I/O for the first buffer before utilizing it to process the next set of records. Our measurements indicate that CPU never had to wait in this phase. The output buffer was always available (I/O completed) when needed. This phase turned out to be CPU bound rather than I/O bound. The only time CPU was idle was during writing of the last 4 MB buffer.

The entire write phase lasted for approximately 2.65 seconds. During this period, disk subsystem was busy for about 2.55 seconds resulting in a sustained write bandwidth of 39 MB/sec. The I/O subsystem could have delivered more bandwidth if more CPU cycles were available. This could be achieved by reducing the operating system overhead in servicing I/O. The approximate breakdown of the CPU time during the write phase is as follows: initializing memory needed for the output buffers and opening the output files is about 0.1 second, initiating, servicing, and checking on I/O is about 1.1 seconds, and sorting of one million keys (already bucket sorted on high order 7 bits) took about 0.35 second, moving one million 100 byte records from the input buffer (random access) to output buffers (sequential access) accounted for about 1 second, and waiting for the last buffer to be written out took about 0.1 second. Now we are 4.85 second into the sort program.

The write phase is followed by the shutdown phase. During this phase, we close all output files and release all the memory back to the system (this takes approximately 0.2 second), and return to AIX command shell.

## 6 Summary Of Results

To summarize, the sort program takes 5.1 seconds from launch to termination. During this period, the I/O subsystem takes 2.1 seconds to read 100 MB of data and 2.55 seconds to write out the output file. There is an overall overhead of 0.45 second where the I/O system is not active. The approximate breakdown of the CPU time is as follows: initiating, servicing, and checking on I/O - 2.2 seconds, initializing and releasing the memory required for the sort application - 1.05 seconds, extraction and sorting of one million keys - 0.6 second, moving of 100 MB of data from input buffer (random access) to output buffer (sequential access) - 1.0 second, and miscellaneous overheads and wait time - 0.25 seconds.

Note that actual sorting of keys takes only a small fraction (12%) of the total CPU time. The rest of the time is taken in servicing I/O, memory, and moving records. By using sort algorithms well suited to RISC super scalar processors, we have reduced the actual sort time to a small fraction. Now the only limiting factor is I/O. If the operating system involvement in servicing I/O and memory can be reduced, then CPU can profitably exploit a higher bandwidth disk system, resulting in an even higher level of performance. The bandwidth is eventually limited by the capacity of the bus connecting the memory system with the disk system. In the near future, we expect faster buses connecting the two sub-systems. Even at present, high end servers have multiple buses connecting the two. Therefore, in the near future, we can expect bandwidths in excess of 100 MB/sec. To actually realize this bandwidth, the operating system has to be made more efficient in handling large block sequential I/O. This is particularly important for decision support systems where high bandwidth is very important. There is no technical reason why this can not be done.

## 7 Bucket Sorting

In bucket sorting, keys are assigned to one of the $k = 2^m$ buckets, based on the high order $m$ bits of the keys. This is a very powerful technique, especially for randomly distributed keys. It avoids $m$ compares per key. This also improves data locality in sorting keys within a bucket, because now each bucket is much smaller and may actually fit in cache. We first used this technique to sort an integer array of size 33 million on an IBM R/S 6000 SP2 scalable parallel computer. This is one of the kernels of an established supercomputer benchmark published by the NAS group of NASA Ames [12]. The key distribution for this benchmark was Gaussian and therefore we modified our implementation to work well on non-uniformly distributed keys [13]-[14]. The sorting is repeated 10 times on integer arrays of size $2^{25}$ (33 million). Since keys are generated as part of the program, this program does not require any I/O. However, it does require rather extensive communication between all nodes of the machine to exchange keys and ranks. On a 64 node SP2, it takes less than four seconds to do the benchmark (less than 0.4 second per sort, as ten sorts are done during the benchmark). By comparison, a Cray T3-D with alpha processors requires four times as many nodes to achieve this level of performance [13]. For SP2, this works out to about 1.25 million keys sorted per node per second. More than half the time was spent in communication. The actual compute time per million keys per node was about 0.32 second. The Datamation benchmark takes approximately 0.6 second (or about 40 cycles per key). The increase is primarily due to the additional cost of extracting keys from 100 byte records. The cost due to longer keys (80 bits vs 21 bits for the NAS benchmark) is minimal.

In implementing a bucket sort, the number of buckets used is limited primarily by cache and TLB considerations. If $k = 2^m$ buckets are used, then during bucket sorting, there are $k$ active memory pointers where data is being written. To avoid cache/TLB thrashing, cache and TLB should have at least $k$ slots. RS/6000 39H has 1024 cache lines of size 128 bytes each and 512 TLB slots for 4K size pages. Based on the TLB considerations and actual measurements, we decided to do bucket sorting on high order 7 bits ($m = 7$), resulting in 128 buckets. In bucket sorting, we stored next 31 bits of the key (after masking off high order 7 bits which have already been sorted in the bucket sorting phase) as an integer word (32 bits). We decided to use only 31 key bits so that we do not have to deal with negative integers. We also stored the location of the record in memory (corresponding to the key), as another 32-bit integer word next to it. Throughout rest of the processing, these two integers (middle key bits and record pointer) are kept together so that when sorting is completed and keys have been arranged in sorted order, we have the location of the corresponding record available next to it. This record location information is eventually used to move records from the input buffer to the output buffer. For the purpose of moving data, these two 32-bit integers can also be treated as a 64-bit floating point number. On many machines, load/stores on 64-bit floating point numbers do not take any longer than load/stores for 32 bit integers. This considerably improves performance. RS/6000 power2 processors also support quad load/store instructions which can load/store 128-bits of data into floating point registers. It can do two such instructions in a cycle. As long as we are only doing copies, it does not matter what type of registers are used. Floating point turns out to be better than fixed point.

In bucket sorting, we are accessing and extracting only 38 (7 + 31) high order bits of the key. However, this requires bringing a 100 byte record into cache. This is where most of the time is spent. However, fortunately records are accessed sequentially. To minimize the cache miss penalty, we implemented a software cache prefetching scheme, whereby, we load (but not actually use it) a four byte data in a register, from the second next record. This results in a cache miss which is serviced concurrently with processing of the current record (key extraction, storing two 32-bit integers in appropriate buckets, update pointers etc.). This software cache pre-fetching is crucial to high performance in memory systems with long latencies [15]. For sequential access, some form of hardware pre-fetching can be implemented which can bring the next set of cache lines. However, for random access, hardware pre-fetching does not work and only software pre-fetching can provide the performance.

As mentioned before, this phase is estimated to take about 0.25 second for one million records on a 66.6 MHz machine. This works out to about 17 cycles per record. The record size is slightly smaller than cache line size of 128 bytes. Almost all the time in this phase is taken in servicing the cache misses. Fortunately, all other processing can be overlapped with it.

Because of the random distribution of keys, every bucket consists of approximately 1,000,000/128 = 7812 (key, pointer) pairs. This occupies 64 KB of memory per bucket. The machine has a 128 KB cache and therefore, two such buffers can fit in cache. This makes further sorting of keys, almost entirely cache resident.

## 8 Radix/Distribution Count Sorting

Let us discuss further sorting which is done one bucket at a time. In radix sorting, you start sorting from low order key bits and move towards high order key bits. To implement pure radix sorting, you will need to sort on all the remaining key bits ($80 - 7 = 73$). However, this is an overkill for random keys. As a rule of thumb, for random keys, the probability of a tie rapidly reduces, once you go past $\log(n)$ key bits where n is the number of keys to be sorted. In our case, $\log(n)$ is 20, and we have already sorted on high order 7 bits. We decided to sort on 22 additional key bits. This will bring total number of key bits sorted $= 7 + 22 = 29$, and it reduces the probability of a tie in 29 bits to be approximately $(2^{-29}) * n = 0.002$. This is a low enough probability that a simple scan and exchange algorithm (described in the next section) will sort the entire sequence at a very low cost. Remember, one of the primary aims in developing this sort algorithm is to avoid compares for RISC processors. In a high performance RISC system with multiple functional units (such as RS/6000 39H), compare and branch sequences are very expensive.

Let us discuss how we sort on the next 22 bits. These are grouped into two pairs of 11 bits each. We set up two count arrays of size $2^{11}$ each. $Count1$ array is used to count on all 11-bit patterns in the low order, and $Count2$ array is used to count on all 11-bit patterns in the high order. One pass through the key array (on a single bucket), provides the count information for $Count1$ as well as $Count2$. These two arrays are integrated (discrete summation) to produce $Rank1$ and $Rank2$ arrays of size $2^{11}$ each. $Rank1(i)$ is the count of keys having their low order 11-bit value less than $i$. $Rank2(i)$ is defined similarly. Next, $Rank1$ array is used to sort on low order bits. The (key, pointer) pair is moved to an $AUX$ array, using the $Rank1$ array value corresponding to 11 low order bits. This is followed by updating the $Rank1$ value to point to the next location in $AUX$ array. This scheme is also called a distribution count sort. It has $2^{11}$ active memory pointers. However, since $AUX$ array fits in cache, there is no performance impact. Floating point values are used to represent the 64-bit (key, pointer) pair. This improves performance in moving data from one array to another array.

The next step is to do similar sorting based on the $Rank2$ array, which corresponds to 11 high order bits. This time, the (key, pointer) pairs are moved back to the original bucket. It can be shown that this results in sorting on 22 high order bits. This has been accomplished without any compares and using instructions which can execute in parallel resulting in a large number of instructions executed per cycle on a processor having multiple execution units. This concept can be extended to sort on any number of bits. Typically, one pass through data can sort 11-12 key bits. At the end of this phase, we have an almost sorted key list. The probability of a tie (on 29 bits sorted so far) is around 0.002. Furthermore, the probability of having multiple ties is even lower. The final sorting to resolve these ties is described in the next section.

## 9 Final Sorting

During this phase, we use all 31 key bits stored in the bucket. If key($i$) is less than key($i+1$), then keys are in the right sequence, if key($i$) equals key($i+1$), then we have to examine the remaining ($80 - 38 = 42$) key bits to decide on their ordering. If key($i$) is greater than key($i+1$), then keys are in the wrong sequence and they have to be exchanged, the $i$ pointer has to be reduced by one, and further comparisons are needed to make sure that they are in the right order. All this requires compare and branch sequences which we want to avoid. Therefore, we work on a section of keys (of size 64, a tunable parameter). For this section, the probability of having any key in a wrong order is approximately $0.002 * 64 = 0.128$. This is low enough for our purposes. We compute differences, key($i$) - key($i+1$), and "AND" them together for a section. During this computing, a high degree of functional parallelism can be exploited, by unrolling the code. If the overall "ANDED" value is negative then all keys are in the right order. Otherwise, we process the section using compares, one element at a time. Note that by reducing the probability of ties, we have eliminated most of the compares. Overall, we have reduced the number of compares per key from $\log(n) = 20$ to approximately 0.128.

## 10 Record Permutation

After a bucket is fully sorted, its (key, pointer) array is used to move records from the input buffer to the output buffer. Pointer($i$) points to the input record which is to be stored in $i$-th position of the output buffer. This requires random access on the input buffer and sequential access on the output buffer. Random access on the input buffer is very expensive. It invariably results in a cache and TLB miss; possibly two misses if the record crosses cache line and/or page boundary. Again, software pre-fetching helps in hiding some of these latencies. While, we are moving the current record, we pre-fetch the next record. Floating point quad load/store instructions are used to copy data. As explained before, a single quad load/store instruction can handle 16 bytes of data and two such instructions can be executed in one cycle. This also improves performance.

This phase is estimated to take about 1.0 second. This works out to about 66 cycles in moving a 100 byte record from input buffer to the output buffer. Most of this time is spent in cache and TLB miss penalties. This penalty could have been reduced if we had implemented a two-stage move. However, this will require moving records twice and overall performance may not improve.

## 11 Summary and Conclusions

In this paper, we have described a sort technique which essentially eliminates all compares and additionally requires very few passes through data. On a desktop IBM RS/6000 39H workstation, it takes only 0.6 second to extract and sort one million keys. The ideas presented in this paper can be generalized to sort variable length keys with skews.

We were able to exploit functional parallelism provided by power2 processor line of RS/6000. We also reduced the number of passes through keys from 20 to 5. All these factors are responsible for very high performance in (key, pointer) sorting. Although our current implementation is for this benchmark, similar ideas can be applied to implement a general sorting routine with variable length binary keys. We have already implemented a routine to sort 32-bit integer keys which gives comparable performance for almost any kind of key skews.

The Datamation sort benchmark sorts a 100 MB disk resident file and stores the sorted output file back to disk. We combined our high performance sort algorithm with a high performance I/O system (IBM SSA disks) to achieve a new performance (for a uni-processor) and price/performance record. Clearly, more work needs to be done to further advance the state of the art. One of the major issue relates to significant operating system overhead in servicing I/O. If this can be improved, even higher level of performance can be achieved.

**Acknowledgments:** This work could not have been done without help and support from many individuals. John Cocke made me aware of this benchmark and the work on Alpha Sort. Steve Watts was instrumental in completing this project. He provided the crucial I/O software and ran the benchmark on various platforms. Mahesh Joshi helped me in installing the I/O software. Eliot Lum provided SSA hardware. Mike Andreasan assembled the benchmark system. Pat Buckland, Steve Furniss, and David Whitworth provided very important performance data and advice on SSA hardware. My manager Fred Gustavson was very supportive of this work and he arranged for all the resources needed to carry out this work. I also benefited from many technical discussions with him and M. Zubair. Clod Barrera, Mike Blasgen, Ashok Chandra, John Forrest, Dan Graham, Ambuj Goyal, James Hamilton, Paul Horn, Bala Iyer, David Jensen, Anant Jhingran, Jai Menon, Alan Petersburg, Bill Pulleyblank, Deepu Rathi, Bernie Rudin, and Irving Wladawsky-Berger provided constant support and encouragement. I had many useful interchanges with Jim Gray and Chris Nyberg that helped me understand issues related to sorting. They also provided a pre-print of their paper. Several other people helped me in various phases of this work.

## References

[1] Anon-Et-Al. "A Measure of Transaction Processing Power", *Datamation*, **V. 31(7)**, pages 112–118, 1985.

[2] Baugsto, B.A.W., Greipsland, J.F. "Parallel Sorting Methods for Large Data Volumes on a Hypercube Database Computer", Proc. 6th Intl. Workshop on Database Machines, Deauville, France, Springer Verlag Lecture Notes No. 368, June 1989, pp. 126-141.

[3] Baugsto, B.A.W., Greipsland, J.F., Kamerbeek, J. "Sorting Large Data Files on POMA", Proc. CONPAR-90 VAPPIV, Springer Verlag Lecture Notes No. 357, Sept. 1990, pp. 536-547.

[4] Cvetanovic, Z., Bhandarkar, D. "Characterization of Alpha AXP Performance using TP and SPEC Workloads", Proc. Int. Symposium on Computer Architecture, April 1994.

[5] DeWitt, D.J., Naughton, J.F., Schneider, D.A. "Parallel Sorting on a Shared Nothing Architecture Using Probabilistic Splitting", Proc. First Int. Conf. on Parallel and Distributed Info Systems, IEEE Press, pp. 280-291, Jan. 1992.

[6] Gray J. (ed.) *The Benchmark Handbook for Database and Transaction Processing Systems*, Morgan Kaufman, San Mateo, 1991. pp. 18-32, 1988.

[7] Knuth, D.E., *Sorting and Searching, The Art of Computer Programming*, Addison Wesley, Reading, MA, 1973.

[8] Tsukerman, A., "FastSort - An External Sort Using Parallel Processing", Proc. SIGMOD 1990, pp. 88-101.

[9] Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., Lomet D., "AlphaSort: A RISC Machine Sort", Proc. SIGMOD 1994, pp. 233-242.

[10] Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., Lomet D., "AlphaSort: A Cache Sensitive Parallel External Sort", To be published in the Proc. of VLDB.

[11] SGI Media Brief dated May 22, 1995.

[12] Bailey, D., Barszcz, E., Barton, J. Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schereiber, R., Simon, H., Venkatakrishnan, V., Weerantunga, S., *The NAS Parallel Benchmarks*, Technical Report RNR-94-007, NASA Ames Research Center, March, 1994.

[13] Agarwal R. C., Gustavson F. G., Zubair M., "A Scalable Parallel Implementation of the NAS Integer Sort Benchmark", Proc. Int. Workshop on Parallel Processing, Bangalore, India, pp. 463-477, December, 1994.

[14] Agarwal R. C., Alpern B., Carter, L., Gustavson, F. G., Klepacki, D. J., Lawrence, R., Zubair, M., "High performance Parallel Implementations of the NAS Kernel Benchmarks on the IBM SP2", *IBM Systems Journal*, **V. 34(2)**, pp. 263-272, 1995.

[15] Agarwal R. C., Gustavson F. G., Zubair M., "Exploiting Functional Parallelism of Power2 to Design High-Performance Numerical Algorithms", *IBM Journal of Research and Development*, **V. 38(5)**, pp. 563-574, 1994.