GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management

Sorting large datasets is often limited by I/O bandwidth in terms of memory and disk. The traditional von Neumann architecture results in high cache misses within L1-L3 cache levels. The authors realized that the highly parallelized processors and the fast memory interconnects inside commodity GPUs can help to work around the limitations that arise when sorting is done solely on the CPU. Specifically, the limitation is that most sorting performance is limited by cache misses. Although main memory sizes have gotten larger, so that disk I/O is reduced, memory access itself is a bottleneck. Utilizing the GPU is advantageous in that it may have 10 times the bandwidth for memory that a CPU has.

In dealing with a large number of records, two pass sorting usually has to be done, because all the data cannot fit into main memory at the same time. The relationship between disk and memory says that memory grows as the square root of the input size, which means terabyte files can be processed with gigabyte main memory. Unfortunately, peak performance is not achieved with traditional CPU based algorithms. There are too many cache misses, peak I/O throughput is never reached and SMT hardware has too much contention for memory between threads. Instead, GPUs can be utilized in non-obvious way. First, they have much more inherent parallelism than SMT CPUs (by using fragment processors and instruction level parallelism with multiple ALUs) and second, the memory interface is designed to have high bandwidth and low latency compared to the CPUs.

The GPUTeraSort algorithm has a number steps which use the strengths of the CPU and the GPU to achieve higher performance. For instance, sorting is done on the GPU so that it can interface with memory quickly, but data loading and key generation is done through the CPU. Writing back to disk is done directly through a DMA. The input to be sorted must be mapped into a form recognizable by the GPU, which operates on 2-D arrays of pixels. A bitonic sequence is thus partitioned and inputted as pixels to the GPU. Calculations are performed which compare the pixels, reorder and merge them. Care needs to be expressed to avoid too many cache misses even in the GPU, so the algorithm uses blocking in order to ensure the data being used is in the same spatial area.

The measured performance is impressive compared to other methods, even noting that more could be gained through hand-tuning. Peak I/O is almost achieved in the CPU by hiding latencies and off-loading work to the GPU. The algorithm has a few limitations: it requires programmable GPUs and it uses radix sort, which means that keys must have a fixed number of bytes), but by using commodity hardware, they have achieved the fastest PennySort result as of yet.
Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting

Sorting in parallel among many nodes requires a means for distributing keys efficiently and effectively. In a shared nothing architecture, messages have to be communicated across a network as a way to coordinate activity. The main function of this communication is to inform nodes of where to send the keys located on its local disks. There are two general methods for determining this information: probabilistic splitting and exact splitting. The paper shows that a probabilistic method is the best performer.

In probabilistic splitting, a random sample is taken of the set and the exact split is found for this smaller subset. It attempts to approximate the overall sample, but is prone to skew if the sample size is too small. On the other hand, if it is too large then finding the exact split is costly. The sorted sample is then sent to a coordinator node which merges all samples and find the exact split of all merged samples. Afterwards, the splitting table is sent to all nodes so that each knows where to send a record. As records arrive, they enter a sort buffer, which when full, is sorted and outputted to disk. All these runs are later merged. The final result is divided, in order, amongst the local disks of all machines.

In exact splitting, each processor completely sorts its fragment on local disk and produces a run for each processor to handle. Then a portion of this sorted set is sent to a coordinator which merges the sets and finds the exact splitting values to be used through a coordinated (and complicated) binary search. A similar process to probabilistic splitting is then followed to sort and merge results.

The authors found that probabilistic splitting performed much better and that dealing with skew from finding imperfect splits was easier than trying to find the perfect splits. In fact, with exact splitting adding more processors tends to make it slower after a threshold since finding a split for each processor is increasingly difficult as the number of processors grow.

The algorithm was implemented on a Gamma Database machine, connected on a fast hypercube network, using an operating system that provides lightweight processes. One artifact of this system that had to be overcome was random seeks and page size. Using the system default page size provided poor performance. Increasing to a larger size helped improve results by 30%.

The results of the implementation showed that taking more random samples reduces skew and thus total time, but only up to the point where the sampling takes too long. They also tested the scaleup ability to see if the system would continue to perform at the same ratio as processors increased (it did not). The speedup was measured to find how much faster the sorting could be done by adding extra processors (it improved, but not perfectly). Finally the sizeup was checked to see how the algorithm behaved when the number of records increased (it achieved sublinear sizeup because random sampling does not depend on the size of the dataset).
High-performance sorting on networks of workstations

Traditionally, the best method for processing data-intensive applications, like sorting, was done through powerful and expensive single node machines. The authors changed this assumption by showing that a network of commodity priced workstations, using standard Unix, could deliver better performance. Moreover, a NOW cluster is incrementally scalable and allows for easy analysis due to the separation of resources.

The authors were interested in trying to surpass the metrics for time to sort a million records and how much data can be sorted in a minute. In their attempts, they used two different clusters. A smaller 8 node system that had larger memory and four disks and a more expansive 64 node system with smaller memory and only two disks. Communication is done through Active Messages, a lightweight RPC protocol designed for LANs.

Four algorithms are presented that were designed for different cluster and dataset characteristics. In the simplest, one-pass single node sorting, data fits completely in the main memory of one computer. In this instance, the actual sorting takes very little time compared to the I/O tasks of reading and writing from the disk. Because they had different kinds of disks, they needed to know how best to stripe data across the disks. Another issue that arose was determining how much memory was available to user programs. Once that limit was overextended, cache misses would dramatically reduce performance. Finally, the authors needed a way to read data from the disk. The commonly used read syscall created double buffers. They settled on using memory mapped files and advising the OS on what blocks should be kept and prefetched. The actual sorting was almost an afterthought and they settled on a bucket sort followed by a partial radix sort, which reduced the time spent sorting to 4% of the total time.

This algorithm allowed for a straightforward evolution to the parallel version. The main difference is that during the reading of keys on each node (keys haven been uniformly distributed prior), the nodes send the keys to the node which will handle it. Nodes perform a barrier synchronization so that all start sorting at the same time, knowing all keys have arrived. With parallel processing there is opportunity for overlapping different aspects of the algorithm to hide latencies of I/O. They chose a combination of a reader thread and a sender thread which can run at the same time on a node. This system scales almost linearly with the number of workstations, although with more nodes the startup time becomes the overwhelming factor.

Because most datasets are too large to fit entirely in main memory, two pass sorting algorithms need to be used. These create a number of runs to operate on in main memory and then write these runs back to disk. The second phase is merging the runs. The size of the runs and the number of runs both affect performance and tradeoffs need to be made whether the first phase or the second phase will take longer respectively. The parallel and single node versions are generally the same except for the sending phase. Interestingly, in the parallel version the parts that should be overlapped are determined by the number of disks. For instance, if there are four disks, reading and writing should be done synchronously instead of being overlapped. A final observation was the limitations of the workstation bus between disk and CPU which made achieving peak performance impossible with more than two disks.