Scalable, Multithreaded, Partially-in-place Sorting

David J. Haglin
and Robert D. Adolf
Pacific Northwest National Laboratory
Richland, Washington 99352, USA
Email: david.haglin@pnnl.gov
Email: robert.adolf@pnnl.gov

Greg E. Mackey
Sandia National Laboratories
Albuquerque, NM 87185, USA
Email: gemacke@sandia.gov

Abstract

A recent trend in hardware development is producing computing systems that are stretching the number of cores and size of shared-memory beyond where most fundamental serial algorithms perform well. The expectation is that this trend will continue. So it makes sense to rethink our fundamental algorithms such as sorting. There are many situations where data that needs to be sorted will actually fit into the shared memory so applications could benefit from an efficient parallel sorting algorithm. When sorting large data (at least hundreds of Gigabytes) in a single shared memory, there are two factors that affect the algorithm choice. First, does the algorithm sort in-place? And second, does the algorithm scale well beyond tens of threads? Surprisingly, existing algorithms possess either one of these factors, but not both. We present an approach that gracefully degrades in performance as the amount of available working memory decreases relative to the size of the input.

1. Introduction

As shared-memory computing systems grow in number of cores and size of memory, it will be increasingly important to have fundamental data structures and algorithms available to application developers in order to effectively take advantage of these systems. Some examples of these large-scale data structures are described in [1], [2], [3].

For our study, we assume a fixed machine (fixed memory size and fixed number of concurrent processors) and examine how the machine performs when sorting integer arrays residing in a significant portion of the memory. Let M be the size of the total memory available to a program and let $|A|$ be the size of the array (in the same units as M). Then, we can use the ratio $\alpha = |A|/M$ to represent the portion of memory required for the data. Note that we assume that $1 - \alpha$ is available for work space during sorting.

Given a fixed M, if our data size produces an $\alpha = 0.5$, then we have many options for sorting algorithms, including *merge sort* and *radix sort*. However, if we have a data size such that $\alpha > 0.5$, then using these algorithms is no longer an option and we must resort to either a completely in-place sorting algorithm such as Batcher's bitonic sort [4], or use an algorithm that requires only limited extra work space.

We might consider *quicksort* [5], which uses only $O(\log n)$ working memory for the recursion bookkeeping. But the parallelization of that algorithm is typically done by recursively sorting the *left* and *right* partitions in parallel with different threads. The partition step is typically done in serial, so the early stages of parallel quicksort are dominated by serial computations. This approach is effective on systems with small numbers (single digits) of threads, but when massive parallelism is required, this approach is ineffective.

This paper proceeds as follows. In section 2 we present a sorting algorithm that can "gracefully" adapt to the amount of available extra memory and in section 3 we discuss other parallel sorting algorithms used in this study. Section 4 describes our experimental process and section 5 presents our findings. Then, a discussion of related work is followed by conclusions.

2. Algorithm Description

Our approach can be thought of as a combination of two algorithms: a parallel version of the partitioning step from quicksort, and a not-in-place sorting algorithm that runs well with massive parallelism. We

call this general strategy the *parallel partitioning sort* (PPsort). These algorithms are composed in such a way that if there is sufficient memory to do the not-in-place sorting, then that is the only algorithm needed. But if there is insufficient memory, the parallel partitioning algorithm is invoked as a pre-processing step to (recursively) arrange the input into smaller, and smaller array sections until the not-in-place sorting algorithm can be used on each of the sections. This strategy is similar to other approaches to accelerate quicksort, but we focus on achieving good performance on massively parallel systems with large shared-memory. Unlike acceleration techniques used for serial quicksort such as invoking an *insertion sort* when the array section becomes small (say 25) [6], we see the partitioning step as purely overhead required by the lack of available working memory, so we stop the recursion as early as possible. We note that by stopping the recursion as early as possible, we avoid the worst-case performance of quicksort ($O(n^2)$) caused by linear depth of recursion due to unfortunate pivot choices.

For our initial exploration we chose the parallel radix sort as our not-in-place sorting algorithm. It would be interesting to try using a parallel merge sort instead, which would produce an overall algorithm that is completely comparison-based.

2.1. Algorithm Overview

A high-level overview of our algorithm is presented in Algorithm 1. The "If" condition at statement 2 can be handled in the C language by doing a `malloc` and checking for a `null` pointer (indicating insufficient memory). The radixSort (called at statement 3) we use is an implementation that runs well with many threads [7]. Note that the PARTITION function called at statement 4 is presented later in Algorithm 2. If there is insufficient free memory available the statements 5 through 8 are essentially the same as a quicksort algorithm, with statement 7 recursively sorting the left partition, and statement 8 recursively sorting the right partition.

As the algorithm proceeds, the recursion may descend several levels until the array section to be sorted is small enough to invoke the not-in-place (radixSort) algorithm. This may not result in recursion of uniform depth, but it does result in calling the not-in-place sorting algorithm as early as possible. This aspect is important because the amount of work to sort each of the pieces is essentially the same amount of work as sorting the entire array (had there been enough working memory), so the partitioning step(s) are all overhead.

Algorithm 1 Parallel Partioning Sort algorithm

```
1:  procedure PPSORT(array, arrayLength)
2:      if sufficient memory exists to allocate a buffer
    to hold arrayLength then
3:          radixSort(array, arrayLength)
4:      else
5:          p ← PARTITION(array, arrayLength)
6:          // recursively sort each part
7:          PPsort(array, p)
8:          PPsort(&array[p+1], arrayLength - p)
9:      end if
10: end procedure
```

2.2. Parallel Partitioning Step

The usual notion of "parallel quicksort" is to do the partitioning step in *serial* and then launch separate threads for each partition. While this may be effective for smaller numbers of threads, the lack of early parallelism with this approach renders parallel performance with hundreds to thousands of processors ineffective.

We consider running the partitioning step in parallel to be mandatory to achieve desired performance. Our approach is based on [8].

1) We first select a pivot value that will be used by all of the threads.
2) We then do a logical striping of the data, one stripe for each thread (see Figure 1).
3) Each stripe is then partitioned by the associated thread using the traditional serial partitioning algorithm [5].

Algorithm 2 Parallel Partitioning Step

```
1:  function PARTITION(array, arrayLength)
2:      pivot ← avg(p values)
3:      for each thread i of p do
4:          c_i ← SerialPartition(array, stripe i, pivot)
5:      end for
6:      l ← min_{1≤i≤p}{c_i}
7:      r ← max_{1≤i≤p}{c_i}
8:      PPSORT(array[l..r], r − l + 1)   ▷ sort middle
9:      m ← binary_search(array[l..r], pivot)
10:     return m
11: end function
```

2.2.1. Pivot Selection. The selection of the pivot in statement 2 is usually done using either: random selection; choosing one of the elements, say the first or last; or picking the median of three (first, middle, and last). Selecting a good pivot is important for quicksort

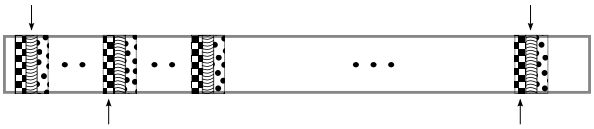

Figure 1. Partitioning layout. Each thread is assigned its own stripe across the data and performs a traditional quicksort partitioning step on its stripe.

to avoid skewed recursion leading to linear depth and therefore $O(n^2)$ running time. In our case the selection of a good pivot—which results in an approximately equal sized left and right partitions—will lead to earlier termination of the recursion and therefore less overhead. Given our assumptions of massive parallelism, we have the opportunity to be more careful on the pivot selection than using the techniques mentioned above.

It turns out that we do not need to select one of the values in the array as our pivot; we merely need some reference point to define (and create) the left and right partitions. Our approach is to select p values from the array at evenly spaced locations and use the mean of those p values as our pivot, where p is the number of threads. Computing this pivot can be done in $O(\log p)$ time.

2.2.2. Striped Partitioning. Each thread has its own stripe of the data that is independent of all other threads' stripes. Consequently, all threads can perform the serial partitioning of their stripe without any need for synchronization. Once all of the threads have completed their partitioning work, they each have a "crossover" point where the pivot would rest within their stripe (see Figure 2). There are three regions of the original array. These regions are demarcated by the smallest and largest thread crossover points. Everything to the left of the smallest crossover point is smaller than the pivot. Everything to the right of the largest crossover point is larger than the pivot. Between the two extremal crossover points is an "unknown" region for which no relation is known between the elements of the region and the pivot. The expectation is that the middle region is quite small and can therefore be sorted using standard techniques (perhaps parallel radix or merge sort) which leaves the final location of the pivot value somewhere in the sorted middle region. The final pivot location can be discovered using a binary search of the middle region. This completes the parallel partitioning step. At this point, all values to the left of the pivot are smaller than the pivot and all values to the right of the pivot are larger than the pivot.

One observation about the middle region is that we

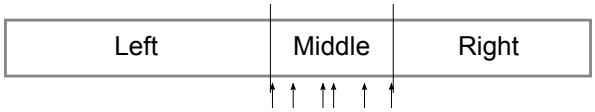

Figure 2. Partitioning layout. When all threads have completed the partitioning of their associated stripe, the crossover points for each thread are noted, and the max and min of these crossover points are determined leaving the original array separated into three sections. The *left* section contains numbers all smaller than the pivot, the *right* section contains numbers all greater than the pivot, and the *middle* section contains numbers with no known relationship to the pivot.

assume there is sufficient working memory to be able to use a not-in-place sorting algorithm on that portion. If the middle region is too large to fit in available working memory, a bitonic sort could be used instead.

2.2.3. Recursion. After finding the location m in statement 9 of Algorithm 2 and returning this location to Algorithm 1, statement 5, we can proceed in manner similar to the traditional quicksort algorithm. That is, we can independently sort the left and right partition in-place resulting in a totally ordered array. Note that the left and right partitions here are not the same as those depicted in Figure 2. The left and right partitions used in the recursion include the appropriate portion of the middle region shown in the figure. Sorting the three parts of Figure 2 (left, middle, and right) independently could result in order violations, as no ordering relationship is known between elements in the left region and elements left of the pivot in the middle region. A similar situation exists for the right region and elements right of the pivot in the middle region. It is necessary to redefine the left and right partitions after finding the m location in Algorithm 2 to include the appropriate elements from the middle region.

3. Other Parallel Sorting Algorithms

We use two parallel sorting algorithms to provide the framework within which we assess our PPsort algorithm. The first is the parallel merge sort included in the *multithreaded graph library* (MTGL) [1]. This parallel merge sort has been optimized for massive parallelism. An open source version of MTGL is available[1]. The MTGL software library is designed to run on the Cray XMT as well as other platforms, but its

1. `https://software.sandia.gov/trac/mtgl`

support of OpenMP is currently limited. We modified the merge sort implementation to insert `OMP` pragmas around the same `for` loops used by the Cray XMT system to run in parallel.

Our second algorithm is a simple bitonic sort as shown in Algorithm 3. The `out_of_order` function at statement 6 must determine, for each call, whether to sort in ascending or descending order. An efficient way to do that is to pass in the value m and compute $x = m$ bitwise AND i. A zero value of x implies ascending order and a non-zero implies descending order. Once the sort order is determined, a check is made to determine if the two array locations are out of order.

Algorithm 3 Bitonic

```
 1: procedure BITONIC(array, arrayLength)
 2:     for (m = 2; m < |A|; m = m × 2) do
 3:         for (r = m/2; r > 0; r = r/2) do
 4:             for (i = 0; i < |A|; i++) do ▷ Parallel
 5:                 j = i bitwise XOR r
 6:                 if out_of_order(a[i], a[j], m) then
 7:                     swap(a[i], a[j])
 8:                 end if
 9:             end for
10:         end for
11:     end for
12: end procedure
```

The outer-most for loop at statement 2 runs $\log n$ times (serially), and the for loop at statement 3 runs between 1 and $\log n$ times, also serially. So the for loop at statement 4 runs $O(\log^2 n)$ times, with a barrier between each run. The inner-most loop can be run in parallel without synchronization.

Consider sorting an array of 32 billion (2^{35}) elements, which has a memory footprint of 256GB. The bitonic sort runs the $O(n)$ inner-loop 630 times. Compare this to a radix sort using an 8-bit "digit" for each phase. This radix sort will run an $O(n)$ operation exactly 8 times to sort an array of 64-bit keys. Moreover, the constant of proportionality on both algorithms is very low, so the radix sort has approximately two orders of magnitude less work to do when sorting data of this size.

4. Experiment Setup

Showing "scalability" is an experimental process that has been evolving recently [9]. As datasets grow in size, and as shared memory, multithreaded systems grow and become more ubiquitous, showing scalability is no longer a matter of showing behavior with processor growth, but focused more on showing behavior on a particular machine as the dataset grows.

Given a fixed system (with a specific shared memory size and number of processors), we vary the amount of data to be sorted. The smallest amount we consider is half of the system memory size, in which case any parallel sorting algorithm that uses n working memory can be used. At the other extreme, if our entire memory is holding the data to be sorted, we have very few options. In our experiments, we sorted this case using a bitonic sort [4]. Since the simplest formulation of a bitonic sort requires that the data be a perfect power of two, all of our experiments used a fixed system memory size that reflects this constraint.

Using terminology presented in section 1, we run our algorithm with varying values of the α ratio for a given memory size M. As a comparison we also run a parallel merge sort on an array of size $M/2$ ($\alpha = 0.5$), and we run a bitonic sort on an array of size M. Our goal is to show that our algorithm can sort data between these 0.5 and 1.0 ratios with graceful degradation between the two extremes.

Each data point is collected by running the same algorithm five times on different datasets and computing an average. We understand that this is not a thorough exploration of our sorting techniques, but the salient point of our approach is the graceful degradation enabled by the parallel partitioning step(s), which is not dependent upon the data as much as it is dependent upon the choice of pivot. Of course, if a significant portion of the data were all the same value, then parallel partitioning performance would be different than on our randomly generated data.

We run a sweep over the α range from 0.5 up to 0.95 for several "fixed" memory sizes on each platform (described later). The fixed memory size is artificially imposed by our testing software. The largest fixed memory size is only half of the physical memory on each platform. This limit is due to the runtime system needing some of the system memory, and a bitonic sort requiring array sizes that are a power of two.

On both of our platforms, we use 64-bit integers as the basic element of the array.

4.1. Platforms

We use two platforms in our experiments that exhibit different properties. The Cray XMT 1 system is typical of a high-end shared memory system with lots of parallelism designed to hide latencies. Commodity x86 systems are emerging into this arena with larger

memories and more cores. Our x86 system has 48 cores and 256GB of shared memory.

4.1.1. Cray XMT1. Cray XMT is the commercial name for the shared-memory multithreaded machine developed by Cray under the code name "Eldorado" [10], [11]. The system is composed of dual-socket Opteron AMD service nodes and custom-designed multithreaded compute nodes with *Threadstorm* processors. The entire system is connected using the Cray Seastar-2.2 high speed interconnect. The system we use in this study has 128 processors and 1 TB of shared memory with 16,384 threads. More details of this architecture are given in [2].

Each Threadstorm processor is able to schedule 128 fine-grained hardware threads (the XMT terminology for this is *stream*) to avoid memory-access generated pipeline stalls on a cycle-by-cycle basis. At runtime, a software thread is mapped to a hardware stream comprised of a program counter, a status word, 8 target registers and 32 general purpose registers. Each Threadstorm processor has a VLIW (Very Long Instruction Word) pipeline containing operations for the Memory functional unit, the Arithmetic unit and the Control unit.

Each Threadstorm is associated with a memory system that can accommodate up to 32GB of 128-bit wide DDR memory. Each memory controller is complemented with a 128KB, 4-way associative data cache to reduce access latencies (this is the only data cache present in the entire memory hierarchy). Memory is structured with full-empty-, pointer forwarding- and trap- bits to support fine grained thread synchronization with little overhead. The memory is hashed at a granularity of 64 bytes and fully accessible through load/store operations to any Threadstorm processor connected to the Seastar-2.2 network, which is configured in a 3D toroidal topology. While memory is completely shared among Threadstorm processors, it is decoupled from the main memory in the AMD Opteron service nodes. Communication between Threadstorm nodes and Opteron nodes is performed through a Lightweight Communication Library (LUC). Continuous random accesses to memory by the Threadstorm processor will top memory bandwidth at around 100M requests per second.

The software environment on the Cray XMT includes a custom, multithreaded operating system for the Threadstorm compute nodes (MTX), a parallelizing C/C++ cross-compiler targeting Threadstorm, a standard Linux 64-bit environment executing on the service and I/O nodes, as well as the necessary libraries to provide communication and interaction between the two parts of the XMT system. The parallelizing C/C++ compiler generates multithreaded code that is mapped to the threaded capabilities of the processors automatically. Parallelism discovery happens fully or semi-automatically by the addition of `pragmas` (directives) to the C/C++ source code. The compiler's parallelism discovery focuses on analyzing loop nests and mapping each loop's iterations in a data-parallel manner to threads.

4.1.2. 48-Core OpenMP Server. Our OpenMP shared memory server integrates 4 AMD Opteron 6176SE processors, code name "Magny Cours". The Opteron 6176SE is a dual die processor that integrates 12 cores (6 per die). Each core has private instruction and data L1 caches of 64 KB each, and a private L2 cache of 512 KB. Each die has a shared cache of 6 MB and two DDR3 channels The two dies in a processor are connected through a HyperTransport coherent connection. The processor operates at a frequency of 2.3 GHz, giving a theoretical memory bandwidth for a 12-core processor of 42.7 GB/. However, the northbridge in each die runs at only 1.8 GHz, limiting the bandwidth to a maximum of 28.8 GB/s. The four processors in our system are connected through HyperTransport links, and the overall system has 256 GB of DDR3 memory at 1333 MHz.

4.2. Datasets

We use the system RANDOM() function to generate all of our data on both platforms. We generate the gigabytes of data for each of the five trials. In many cases, our platform spent more time generating the data than it spent sorting the data. The Cray XMT platform has special library functions for generating lots of random numbers in parallel using all of the available threads.

5. Experiment Results

On each of our two platforms we artificially imposed memory limits of several sizes up to half the size of the physical memory. For each of these scenarios, we ran a merge sort on half of the artificial limit, did a parameter sweep on the ratio α on PPsort, for $\alpha = 0.5, 0.6, 0.7, 0.8, 0.9,$ and 0.95. Finally, we ran bitonic sort on the full amount of the artificial limit.

5.1. Cray XMT

The Cray XMT has 128 processors and 1024GB of shared memory. We ran five sets of experiments on this machine with fixed memory sizes of 512GB, 256GB,

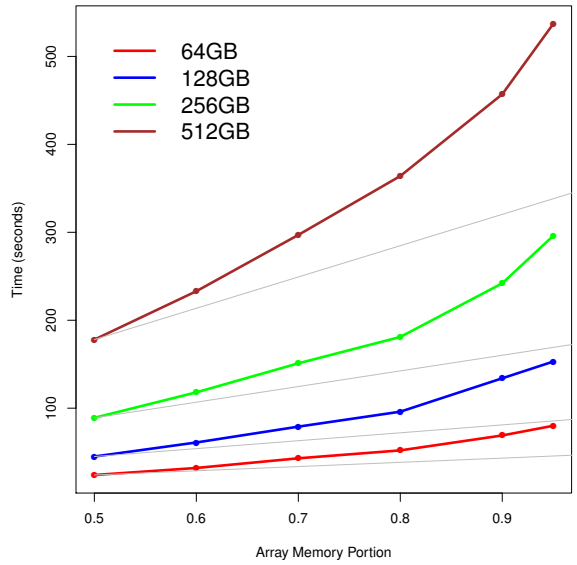

Figure 3. Results on the Cray XMT1. Each curve shows performance on a fixed memory capacity while varying the size of the array to be sorted in that memory. The x-axis represents the ratio of the size of the array to be sorted to the size of the machine. For example, the red line shows the performance on a 64GB machine. At the 0.6 axis point, the red line indicates the time required to sort a 38GB array on a 64GB memory space (so there are only 26GB of available memory). The y-axis indicates time in seconds.

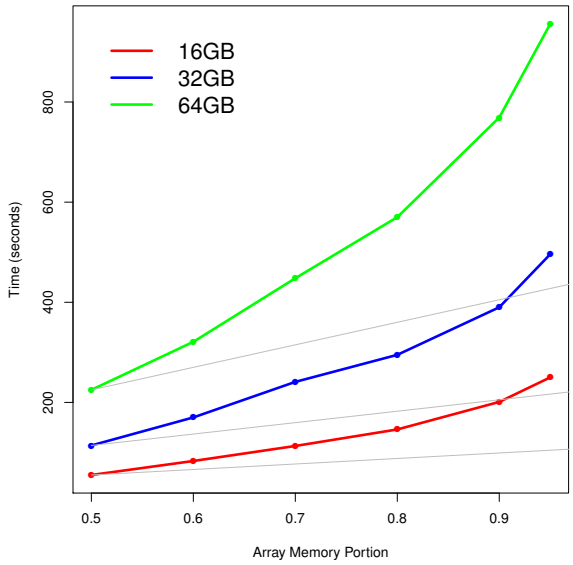

Figure 4. Results on an OpenMP shared memory server. Each curve shows performance on a fixed memory capacity while varying the size of the array to be sorted in that memory. The x-axis represents the ratio of the size of the array to be sorted to the size of the machine. For example, the red line shows the performance on a 64GB machine. At the 0.6 axis point, the green line indicates the time required to sort a 38GB array on a 64GB memory space (so there are only 26GB of available memory). The y-axis indicates time in seconds.

128GB, 64GB, and 32GB. Table 1 shows the results for the five fixed memory sizes on the parallel merge sort, PPsort on various α values, and the bitonic sort. For convenience, we placed the size of the array to be sorted underneath the running time in seconds.

The plot in Figure 3 shows only the PPsort algorithm performance. Including the bitonic performance in this plot would have changed the scale too much. Note that the gray line starting from each of the plot points at $\alpha = 0.5$ shows a linear increase in running time relative to the size of the array. This provides a comparison between an optimistic linear growth and actual performance of PPsort. We assume that the radix sort is essentially linear in growth, so the difference between the PPsort performance and the gray "ideal" is due to the overhead of the partitioning step(s).

5.2. OpenMP

The OpenMP server has 48 cores and 256GB of shared memory. We ran four sets of experiments on this machine with fixed memory sizes of 128GB, 64GB, 32GB, and 16GB. Similar to our Cray XMT results, Table 2 shows the results for the four fixed memory sizes on the parallel merge sort, PPsort on various α values, and the bitonic sort. For convenience, we placed the size of the array to be sorted underneath the running time in seconds.

As in the plot for the Cray XMT results, we include only PPsort times in the plot for the OpenMP server in Figure 4. It is interesting to compare the raw times for the memory sizes that appear in the results of both platforms.

We notice that the overhead cost on the OpenMP server seems to be higher than on the Cray XMT. The curves rise faster on the OpenMP results, where the times reported at the $\alpha = 0.95$ ratio are higher than the next larger memory size ideal gray line. Whereas, the times reported on the Cray XMT results at the $\alpha = 0.95$ ratio remain below the next larger memory size ideal gray line.

Table 1. This table shows runs on a Cray XMT1 with 128 processors. All times are shown in seconds. For convenience, the size of the array to be sorted is shown underneath the row of times for each of the machine memory sizes. As an example, if the machine size is 32GB and the α ratio is 0.6, then the size of the array to be sorted is 32GB$\times 0.6 \approx$19GB.

Memory	MergeSort	0.5	0.6	0.7	0.8	0.9	0.95	Bitonic
32GB	22	13	18	25	28	36	43	1178
Array Size	16GB	16GB	19GB	22GB	26GB	29GB	30GB	32GB
64GB	45	24	32	43	52	69	80	1882
Array Size	32GB	32GB	38GB	45GB	51GB	58GB	61GB	64GB
128GB	95	45	61	79	96	134	153	3992
Array Size	64GB	64GB	77GB	90GB	102GB	115GB	122GB	128GB
256GB	191	89	118	151	181	242	296	8454
Array Size	128GB	128GB	154GB	179GB	205GB	230GB	243GB	256GB
512GB	404	178	233	297	364	457	537	17875
Array Size	256GB	256GB	307GB	358GB	410GB	461GB	486GB	512GB

Table 2. This table shows runs on an OpenMP server with 48 cores and 256 GB of shared memory. All times are shown in seconds. For convenience, the size of the array to be sorted is shown underneath the row of times for each of the machine memory sizes. As an example, if the machine size is 32GB and the α ratio is 0.6, then the size of the array to be sorted is 32GB$\times 0.6 \approx$19GB.

Memory Size	MergeSort	0.5	0.6	0.7	0.8	0.9	0.95	Bitonic
16GB	65	55	83	113	146	201	250	1626
Array Size	8GB	8GB	10GB	11GB	13GB	14GB	15GB	16GB
32GB	142	114	170	241	295	390	496	3333
Array Size	16GB	16GB	19GB	22GB	26GB	29GB	30GB	32GB
64GB	238	225	321	448	570	769	956	3800
Array Size	32GB	32GB	38GB	45GB	51GB	58GB	61GB	64GB
128GB	336	366	604	837	1020	1382	1873	4097
Array Size	64GB	64GB	77GB	90GB	102GB	115GB	122GB	128GB

6. Related Work

Sorting is one of the most studied problems in the computing literature. Here we highlight some related works most closely aligned with our work and have ignored the vast majority of the research in sorting due to scope and space limitations.

6.1. Sorting in-place

Franceschini *et al*. present an in-place radix sort algorithm that compresses part of the input array (modifying the keys to be sorted) in order to free up some space to use as work space during the sort [12]. Their algorithm is both in-place and stable, but it is difficult to imagine how to run it in parallel. Other in-

place radix sorting algorithms have also been presented [13], [14], neither of which are conducive to running with massive parallelism.

Arne Maus explores an adaptive radix sort that remains stable even while trading speed for extra space at runtime [15]. This algorithm is presented as a serial algorithm with no discussion about running on a parallel machine. Moreover, the sorting experiments were done on arrays of size less than 1GB (97 million entries), which is less than 2^{27}. Contrast this with our study exploring data whose sizes are 2^{31} to 2^{35}.

6.2. Other results

There are many sorting algorithms developed for distributed memory architectures (e.g., [14], [16], [17], [18]), a problem we consider significantly different from our problem explored here.

There have been many studies, even recently, whose data size is so much smaller than our study that we consider it a different problem set. For example, Biggar *et al.*[19] explore the impact of branch prediction on the instruction pipeline within a sorting algorithm. They consider problem sizes of 2^{12} up to 2^{22} keys. Süß present various strategies for implementing parallel quicksort using OpenMP [20]. In that study, 100 million keys were sorted. Peters *et al.* present an adaptation of the bitonic sorting algorithm for a CUDA based architecture [21]. The experiments were run on data sizes ranging from 2^{10} up to 2^{24} keys

Our experiments were run on data of sizes 16GB on up to 512GB, and each array entry is a 64-bit value. Thus, our arrays are 2 billion (2^{31}) on up to 64 billion (2^{35}) in length. We also ran on platforms with 48 to 128 cores running. In the case of the 128 threadstorm-core XMT, we ran upwards of 14,000 threads.

7. Conclusion

We have presented an algorithmic strategy for supporting the sorting of data on a multithreaded platform using an efficient parallel algorithm where the size of the data is larger than half of the total memory. The cost of going to some other strategy such as a bitonic or even an out-of-memory sorting algorithm can be avoided with relatively little impact.

We note that, on our OpenMP server system, sorting 64GB of data was faster using the parallel merge sort than using the radix sort (see the 128GB row in Table 2). It would be interesting to explore using a parallel merge sort as the not-in-place sorting algorithm, which would produce an overall algorithm that is completely comparison-based.

Acknowledgment

A part of this work was funded by the Center for Adaptive Super Computing Software - MultiThreaded Architectures (CASS-MT) at the U.S. Department of Energy's Pacific Northwest National Laboratory. Pacific Northwest National Laboratory is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000. This paper has a Sandia report number of 2013-4373 C.

References

[1] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny, "Software and algorithms for graph queries on multithreaded architectures," *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 495, 2007.

[2] E. L. Goodman, D. J. Haglin, C. Scherrer, D. Chavarria-Miranda, J. Mogill, and J. Feo, "Hashing strategies for the Cray XMT," in *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, Apr. 2010.

[3] E. Goodman, M. N. Lemaster, and E. Jimenez, "Scalable hashing for shared memory supercomputers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 41:1–41:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063439

[4] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, ser. AFIPS '68 (Spring). New York, NY, USA: ACM, 1968, pp. 307–314. [Online]. Available: http://doi.acm.org/10.1145/1468075.1468121

[5] C. A. R. Hoare, "Quicksort," *Comput. J.*, vol. 5, no. 1, pp. 10–15, 1962.

[6] R. Sedgewick, "Implementing quicksort programs," *Commun. ACM*, vol. 21, no. 10, pp. 847–857, Oct. 1978. [Online]. Available: http://doi.acm.org/10.1145/359619.359631

[7] P. Briggs, "Examples: A working document," Jan. 2011, a collection of illustrative programming examples for the Tera MTA.

[8] S. Kahan and L. Ruzzo, "Parallel quicksand: Sorting on the sequent," Department of Computer Science, University of Washington, Tech. Rep. 91–01–01, Jan. 1991.

[9] J. Weaver, "A scalability metric for parallel computations on large, growing datasets (like the web)," in *Proceedings ofthe Joint Workshop on Scalable and High-Performance Semantic Web Systems*, 2012, pp. 91–96.

[10] D. Chavarría-Miranda, A. Marquez, J. Nieplocha, K. Maschhoff, and C. Scherrer, "Early Experience with Out-of-Core Applications on the Cray XMT," in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, April 2008, pp. 1–8.

[11] J. Feo, D. Harper, S. Kahan, and P. Konecny, "ELDORADO," in *CF '05: Proceedings of the 2nd conference on Computing frontiers*. New York, NY, USA: ACM, 2005, pp. 28–34.

[12] G. Franceschini, S. Muthukrishnan, and M. Pătraşcu, "Radix sorting with no extra space," in *Proceedings of the 15th annual European conference on Algorithms*, ser. ESA'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 194–205. [Online]. Available: http://dl.acm.org/citation.cfm?id=1778580.1778601

[13] A. Al-Badarneh and F. El-Aker, "Efficient adaptive in-place radix sorting," *Informatica*, vol. 15, no. 3, pp. 295–302, Aug. 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413748.1413749

[14] S. Q. Zheng, B. Calidas, and Y. Zhang, "An efficient general in-place parallel sorting scheme," *J. Supercomputing*, vol. 14, no. 1, pp. 5–17, Jul. 1999. [Online]. Available: http://dx.doi.org/10.1023/A:1008173729055

[15] A. Maus, "Buffered adaptive radix – a fast, stable sorting algorithm that trades speed for space at runtime when needed," in *NIK'2007, Norwegian Informatics Conference*, 2007.

[16] A. Datta, S. Soundaralakshmi, and R. Owens, "Fast sorting algorithms on a linear array with a reconfigurable pipelined bus system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 212–222, Mar. 2002. [Online]. Available: http://dx.doi.org/10.1109/71.993203

[17] J. Brest, A. Vreže, and V. Žumer, "A sorting algorithm on a pc cluster," in *Proceedings of the 2000 ACM symposium on Applied computing - Volume 2*, ser. SAC '00. New York, NY, USA: ACM, 2000, pp. 710–715. [Online]. Available: http://doi.acm.org/10.1145/338407.338549

[18] A. S. Arefin and M. A. Hasan, "An improvement of bitonic sorting for parallel computing," in *Proceedings of the 9th WSEAS International Conference on Computers*, ser. ICCOMP'05. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2005, pp. 15:1–15:4. [Online]. Available: http://dl.acm.org/citation.cfm?id=1369599.1369614

[19] P. Biggar, N. Nash, K. Williams, and D. Gregg, "An experimental study of sorting and branch prediction," *J. Exp. Algorithmics*, vol. 12, pp. 1.8:1–1.8:39, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1227161.1370599

[20] M. Süß and C. Leopold, "A users experience with parallel sorting and openmp," in *In Proceedings of the Sixth Workshop on OpenMP (EWOMP04)*, 2004.

[21] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, "Fast in-place sorting with cuda based on bitonic sort," in *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, ser. PPAM'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 403–410. [Online]. Available: http://dl.acm.org/citation.cfm?id=1882792.1882841