Highly Scalable Parallel Sorting

Edgar Solomonik and Laxmikant V. Kalé

Department of Computer Science
University of Illinois at Urbana-Champaign,
Urbana, IL 61801, USA
E-mail: solomon2@illinois.edu,
kale@illinois.edu

***Abstract*— Sorting is a commonly used process with a wide breadth of applications in the high performance computing field. Early research in parallel processing has provided us with comprehensive analysis and theory for parallel sorting algorithms. However, modern supercomputers have advanced rapidly in size and changed significantly in architecture, forcing new adaptations to these algorithms. To fully utilize the potential of highly parallel machines, tens of thousands of processors are used. Efficiently scaling parallel sorting on machines of this magnitude is inhibited by the communication-intensive problem of migrating large amounts of data between processors. The challenge is to design a highly scalable sorting algorithm that uses minimal communication, maximizes overlap between computation and communication, and uses memory efficiently. This paper presents a scalable extension of the Histogram Sorting method, making fundamental modifications to the original algorithm in order to minimize message contention and exploit overlap. We implement Histogram Sort, Sample Sort, and Radix Sort in CHARM++ and compare their performance. The choice of algorithm as well as the importance of the optimizations is validated by performance tests on two predominant modern supercomputer architectures: XT4 at ORNL (Jaguar) and Blue Gene/P at ANL (Intrepid).**

I. INTRODUCTION

Algorithms for sorting are some of the most well-studied and optimized algorithms in computer science. Parallel sorting techniques, despite being well analyzed in a vast amount of literature, have not been sufficiently tested or optimized on modern high performance supercomputers. Good scaling and performance of parallel sorting is necessary for any parallel application that depends on sorting. ChaNGa [1], an n-body Barnes-Hut tree-based gravity code [2], is an example of a highly parallel scientific code that uses sorting every iteration.

The problem posed by parallel sorting is significantly different and more challenging than the corresponding sequential one. *We begin with an unknown, unsorted distribution of n keys over p processors.* The algorithm has to sort and move the keys to the appropriate processor so that they are in globally sorted order. A globally sorted order implies that every key on processor k is larger than every key on processor $k-1$. Further, at the end of execution, the number of keys stored on any processor should not be larger than some threshold value $\frac{n}{p} + t_{thresh}$ (dictated by the memory availability or desired final distribution).

We will describe a new algorithm grounded on the basic principles of Histogram Sort [3] that can perform and scale well on modern supercomputer architectures. Using our optimized implementation, we are able to achieve an estimated 46.2% efficiency on 32,768 cores of BG/P (which corresponds to a speedup of roughly 15,000). For a communication intensive algorithm such as sorting, this is a fairly impressive efficiency. Additionally, the algorithm handles uniform as well as highly non-uniform distributions of keys effectively.

Section 2 describes various parallel sorting algorithms as well as the original Histogram Sort. Section 3 documents the optimizations we applied to the algorithm in order to achieve better scaling. These include our probe determination algorithm, all-to-all optimizations, and exploitations of overlap of communication with local sorting and merging. Section 4 introduces the specifics of our implementation, describes the machines we use for testing, and analyzes the scaling and comparative speed of our parallel sorting algorithm. Section 5 will suggest conclusions that can be drawn from our work as well as potential areas for improvement.

II. PREVIOUS WORK

Previous work has produced many alternative solutions to address the problem of parallel sorting. The majority of parallel sorting algorithms can be classified as either *merge-based* or *splitter-based*. *Merge-based* parallel sorting algorithms rely on merging data between pairs of processors. Such algorithms generally achieve a globally sorted order by constructing a sorting network between processors, which facilitates the necessary sequence of merges. *Splitter-based* parallel sorting algorithms aim to define a vector of *splitters* that subdivides the data into p approximately equal-sized sections. Each of the $p-1$ *splitters* is simply a key-value within the dataset. Thus for splitter s_i, where $i \in \{0, 1, 2, ..., p-2\}$, $loc(s_i) \in 0, 1, 2, ..., n$ is the total number of keys smaller than s_i. Splitter s_i is valid if, $(i+1)\frac{n}{p} - \frac{1}{2}t_{thresh} < loc(s_i) < (i+1)\frac{n}{p} + \frac{1}{2}t_{thresh}$. Once these splitters are determined, each processor can send the appropriate portions of its data directly to the destination processors, which results in one round of all-to-all communication. Notably, such splitter-based algorithms utilize minimal data movement, since the data associated with each key only moves once.

Merge-based parallel sorting algorithms have been extensively studied, though largely in the context of sorting networks, which consider $n/p \approx 1$. For $n/p \gg 1$, these algorithms begin to suffer from heavy use of communication and difficulty of load balancing. Therefore, splitter-based algorithms have been the primary choice on most modern machines due to their minimal communication attribute and scalability. However, newer and much larger architectures have changed the problem statement further. Therefore, traditional approaches, including splitter-based sorting algorithms, require re-evaluation and improvement. We will briefly detail some of these methods below.

A. Bitonic Sort

Bitonic Sort, a merge-based algorithm, was one of the earliest procedures for parallel sorting. It was introduced in 1968 by Batcher [4]. The basic idea behind Bitonic Sort is to sort the data by merging *bitonic sequences*. A *bitonic sequence* increases monotonically then decreases monotonically. For $n/p = 1$, $\Theta(\lg n)$ merges are required, with each merge having a cost of $\Theta(\lg n)$. The composed running time of Bitonic Sort for $n/p = 1$ is $\Theta(\lg^2 n)$. Bitonic Sort can be generalized for $n/p > 1$, with a complexity of $\Theta(n \lg^2 n)$. Adaptive Bitonic Sorting, a modification of Bitonic Sort, avoids unnecessary comparisons, which results in an improved, optimal complexity of $\Theta(n \lg n)$ [5].

Unfortunately, each step of Bitonic Sort requires movement of data between pairs of processors. Like most merge-based algorithms, Bitonic Sort can perform very well when n/p (where n is the total number of keys and p is the number of processors) is small, since it operates in-place and effectively combines messages. On the other hand, its performance quickly degrades as n/p becomes large, which is the much more realistic scenario for typical scientific applications. The major drawback of Bitonic Sort on modern architectures is that it moves the data $\Theta(\lg p)$ times, which turns into costly bottleneck if we scale to higher problem sizes or a larger number of processors. Since this algorithm is old and very well-studied, we will not go into any deep analysis or testing of it. Nevertheless, Bitonic Sort has laid groundwork for much of parallel sorting research and continues to influence modern algorithms. One good comparative study of this algorithm has been documented by Blelloch *et al.* [6].

B. Sample Sort

Sample Sort is a popular and widely analyzed splitter-based method for parallel sorting [7], [8]. This algorithm acquires a sample of data of size s from each processor, then combines the samples on a single processor. This processor then produces $p-1$ splitters from the sp-sized combined sample and broadcasts them to all other processors. The splitters allow each processor to send each key to the correct final destination immediately. Some implementations of Sample Sort also perform localized load balancing between neighboring processors after the all-to-all.

1) Sorting by Regular Sampling: The Sorting by Regular Sampling technique is a reliable and practical variation of Sample Sort that uses a sample size of $s = p - 1$ [9]. The algorithm is simple and executes as follows.

1) Each processor sorts its local data.
2) Each processor selects a sample vector of size $p-1$ from its local data. The kth element of the sample vector is element $\frac{n}{p} \times \frac{k+1}{p}$ of the local data.
3) The samples are sent to and merged on processor 0, producing a combined sorted sample of size $p(p-1)$.
4) Processor 0 defines and broadcasts a vector of $p-1$ splitters with the kth splitter as element $p(k+\frac{1}{2})$ of the combined sorted sample.
5) Each processor sends its local data to the appropriate destination processors, as defined by the splitters, in *one* round of all-to-all communication.
6) Each processor merges the data chunks it receives.

It has been shown that if n is sufficiently large, no processor will end up with more than $\frac{2n}{p}$ keys [9]. In practice, the algorithm often achieves almost perfect load balance.

Though attractive for its simplicity, this approach is problematic for scaling beyond a few thousand processors on modern machines. The algorithm requires that a combined sample $p(p-1)$ keys be merged on one processor, which becomes an unachievable task since it demands $\Theta(p^2)$ memory and work on a single processor. For example, on 16,384 processors, the combined sample of 64-bit keys would require 16 GB of memory. Despite this limitation, due to the popularity of Sorting by Regular Sampling, we provide performance results of our implementation of the algorithm (see Figure 7). For a more thorough description and theoretical analysis of Sample Sort, refer to Li *et al.* [10].

2) Sorting by Random Sampling: Sorting by Random Sampling is an interesting alternative to Regular Sampling [8], [11], [6]. The main difference between the two sampling techniques is that a random sample is flexible in size and collected randomly from each processor's local data rather than as a regularly spaced sample. The advantage of Sorting by Random Sampling is that often sufficient load balance can be achieved for $s < p$, which allows for potentially better scaling. Additionally, a random sample can be retrieved before

sorting local data, which allows for overlap between sorting and splitter calculation. However, the technique is not wholly reliable and can result in severe load imbalance, especially on a larger amount of processors. Since Sorting by Random Sampling is marginally different from Sorting by Regular Sampling, we do not present results for this version of Sample Sort.

C. Radix Sort

Radix Sort is a sorting method that uses the binary representation of keys to migrate them to the appropriate bucket in a series of steps. During every step, the algorithm puts every key in a bucket corresponding to the value of some subset of the key's bits. A k-bit radix sort looks at k bits every iteration. Thus a 16-bit radix on 64-bit keys would take 4 steps and use 2^{16} buckets every step. The algorithm correctly sorts the keys by starting with the less significant bits of the keys and moving the keys out of the lower indexed buckets first.

Radix Sort can be parallelized simply by assigning some subset of buckets to each processor [6], [12]. In addition, it can deal with uneven distributions efficiently by assigning a varying number of buckets to all processors every step. This number can be determined by having each processor count how many of its keys will go to each bucket, then summing up these histograms with a reduction. Once a processor receives the combined histogram, it can adaptively assign buckets to processors.

Radix Sort is not a comparison-based sort. However, it is a reasonable assumption to equate a comparison operation to a bit manipulation, since both are likely to be dominated by the memory access time. Nevertheless, Radix Sort is not bound by $\Theta(\frac{n \lg n}{p})$, as any comparison-based parallel sorting algorithm would be. In fact, this algorithm's complexity varies linearly with n. The performance of the sort can be expressed as $\Theta(\frac{bn}{p})$, where b is the number of bits in a key [12]. This expression is evident in that doubling the number of bits in the keys entails doubling the number of iterations of Radix Sort.

The main drawback to *parallel* Radix Sort is that it requires multiple iterations of costly all-to-all data exchanges. The cache efficiency of this algorithm can also be comparatively weak. In a comparison-based sorting algorithm, we generally deal with contiguous allocations of keys. During sequential sorting (specifically in the partitioning phase of Quicksort), we iterate through keys with only two iterators and swap them between two already accessed locations. Communication in comparison-based sorting is also cache efficient because we can usually copy sorted blocks into messages. In Radix Sort, at every iteration any given key might be moved to any bucket (there are 64 thousand of these for a 16-bit radix), completely independent of the destination of the previously indexed key. However, Thearling et al. [12] propose a clever scheme for improving the cache efficiency during the counting stage.

Nevertheless, Radix Sort is a simple and commonly accepted approach to parallel sorting. Therefore, despite its limitations, we implemented Radix Sort and analyzed its performance.

D. Histogram Sort

Histogram Sort is another splitter-based method for parallel sorting [3]. The extension and optimization of this approach will be the focus of this paper. We describe the original formulation of this algorithm here.

Like Sample Sort, Histogram Sort also determines a set of $p-1$ splitters to divide the keys into p evenly sized sections. However, it achieves this task more accurately by taking an iterative approach rather than simply collecting one big sample. The procedure begins by broadcasting k (where $k \geq p-1$) splitter guesses, which we call a probe, from the initiating processor to every other processor. These initial guesses are usually spaced evenly over the data range. Once the probe is received, the following steps are performed.

1) Each processor sorts its local data.
2) Each processor determines how many of its keys fit into every range produced by the guesses, creating a histogram.
3) A reduction sums up these histograms from every processor. Then one processor analyzes the data sequentially to see which splitter guesses were satisfactory (fell within $\frac{1}{2}t_{thresh}$ of the ideal splitter location).
4) If there are any unsatisfied splitters, a new probe is generated sequentially then broadcasted to all processors, returning to step 2. If all splitters have been satisfied, we continue to the next step.
5) The desired splitting of data has been achieved, so the $p-1$ finalized splitters and the numbers of keys expected to end up on every processor are broadcasted.
6) Each processor sends its local data to the appropriate destination processors, as defined by the splitters, in *one* round of all-to-all communication.
7) Each processor merges the data chunks it receives.

We used Histogram Sort as the basis for our work because it has the essential quality of only moving the actual data once, combined with an efficient method for dealing with uneven distributions. In fact, Histogram Sort is unique in its ability to reliably achieve a defined level of load balance. Therefore, we decided this algorithm is a theoretically well suited base for scaling

sorting on modern architectures. Throughout this paper, we will present adaptations and optimizations to this algorithm in order to improve its performance, but we will rely on the skeleton description presented above.

III. Optimizations

The performance of Histogram Sort can be improved by techniques that reduce the number of histogramming iterations needed, overlap communication with computation whenever possible, and minimize network contention. We present a series of optimizations aimed at satisfying these goals and eliminating scaling bottlenecks.

A. Probe Determination

The probe determination algorithm needs to be efficient since it is only executed by one processor, which gives it a high potential of becoming a bottleneck. The desire for quick convergence to the correct splitter locations is counterbalanced by the necessity to keep the splitter guess generation logic simple and quick. Additionally, we aim to design and test a general method that can handle any distribution.

We generally set aside one processor to determine the probe. This processor keeps track of bounds for all the desired splitters. The ideal location (the number of total keys smaller than the splitter) of the ith splitter is $\frac{(i+1)n}{p}$. Whenever a histogram arrives (as in step 3 of Histogram Sort), we iterate through the results and update the desired splitter information to either narrow the range for the next guess or declare the splitter as achieved. By declaring a splitter as achieved, we mean that a splitter guess has partitioned the data within a satisfactory threshold ($\frac{1}{2}t_{thresh}$) from the target location of the splitter.

The size of the generated probe varies depending on how many splitters have yet to be resolved, but we hold k, as previously defined, to be an upper bound for the number of new guesses. If the distribution is uneven, it is very likely that many desired splitters will end up being bound within the same interval (their ideal locations being in between the locations of the same two previous splitter guesses). We take a basic approach and subdivide any interval containing s unachieved splitters with $\lfloor s \times \frac{k}{u} \rfloor$ guesses, where u is the total number of unachieved splitters. We do not spend time interpolating the data. Instead, we distribute the guesses evenly over the interval. This method takes minimal time to generate a probe, while guaranteeing convergence. An interpolation technique decreases the average time for achieving each splitter, but usually increases the time until the last splitter is achieved.

This paper will consider a splitter guess satisfactory if it falls within some threshold of the ideal location of the splitter. Unless stated otherwise, it can be assumed that $t_{thresh} = \frac{n}{10p}$ (10%). This threshold means each splitter is achieved if a splitter guess location is within $\frac{n}{20p}$ of the splitter's ideal location. We do not try to attain exact splitting of data (setting a threshold of zero) because we are not aware of any practical applications that demand this criterion. One method for exact splitting is proposed by Cheng *et al.* [13].

If p is large, we reserve a single processor for the probe generation work. This choice increases the amount of data on each other processor by n/p^2. However, as long as the work necessitated by this extra data is smaller than the probe generation cost, offloading the probe analysis to a single processor reduces the critical path.

B. Sort and Histogram Overlap

From a scalability perspective, an obvious drawback of the original Histogram Sort algorithm is that during the reduction and probe generation period, all processors except one are idle. Since the reduction as well as the probe analysis and production operate with data of size p, the overhead of this stage will inevitably increase as we attempt to use the algorithm on larger machines.

Most approaches to comparison-based parallel sorting begin by sorting the local data first. However, we noticed that Histogram Sort presents an opportunity to overlap communication with computation by sorting while the reduction happens. Rather than sort first and spend time searching for every splitter guess afterward, we determine the locations of guesses within the local data by treating them as if they were Quicksort *pivots* and performing one iteration of Quicksort. An iteration of Quicksort swaps keys until all of the keys smaller than the pivot are separated from the ones larger than the pivot. We will refer to this task as *splicing the data*. This process is implemented by iterating from the start of the array and from the end of the array until the two iterators meet somewhere in the middle. The point at which the two iterators collide is the point where the pivot belongs. Because our pivots are actually splitter guesses, every iteration of Quicksort can effectively determine where the guess fits in the local data. Thus, without having to first fully sort the keys, we can efficiently identify the locations of the guesses.

Now, each processor can utilize the reduction and probe generation time for sorting the local data in pieces. If additional probes need evaluation, we can search for any guesses that are in the range of the sorted data and calculate where the rest of the guesses fit by again splicing the data. Figure 1 might help visualize this process by providing a time line of the flow of events on a few processors. Every line demonstrates

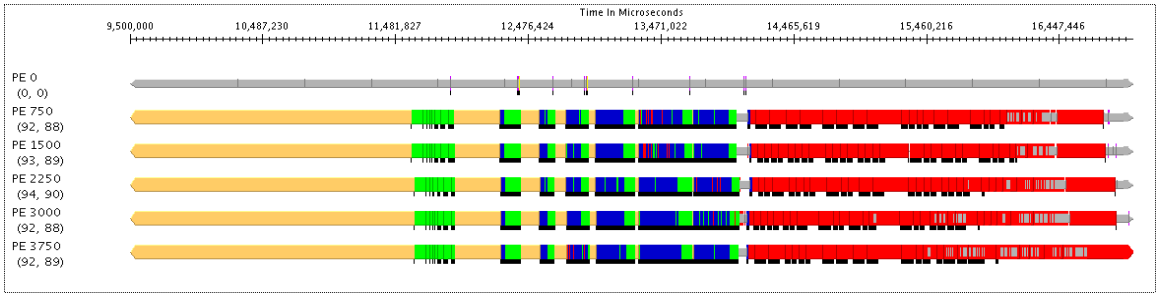

Fig. 1: Timeline of Histogram Sort using the Projections tool. This run was done on Intrepid with 8 million 64-bit keys per core on 4096 cores. The orange (first color in order of appearance) is the probe fitting/splicing time. The green (second color) is the local sorting time. The blue (third color) is a combination of local sorting and sending of data. The red (last color) is the merging work.

the periods of time spent performing each task on one processor. This time line also reflects some features of our implementation that have yet to be explained.

Since Quicksort is widely considered an optimal sequential sort, our procedure determines the location of any splitter guess with only a small overhead in choosing a pivot. This strategy shortens the critical path of the algorithm because the sequential probe determination time and the reduction are now happening concurrently with the local sorting. The effects of this adaptation become more noticeable when scaling the sort since the probe determination and reduction grow in cost.

We also implemented a few optimizations to this approach in order to efficiently sort highly non-uniform datasets. We found that simply selecting the middle splitter guess as a pivot was insufficient. We could not rely on the guess being close to the median of the data, which is an important efficiency factor for Quicksort. To solve this problem, we utilized a few concepts from the Introsort sequential sorting technique, which is used by the C++ Standard Library [14]. The Introsort algorithm uses Quicksort until the subdivisions are small and then finishes with a Heapsort. For Quicksort, it employs the median-of-3 pivot selection, which looks at the first, middle, and last elements of an array to select the pivot. We extrapolated this approach to selecting even larger samples of keys to find a good median. We focused on dealing with comparatively large array sizes (8 million keys), so the cost of finding a median of a larger sample is small compared to the splicing cost. Then we searched the probe to find a guess closest to that median. Splicing small ranges also turned out to be inefficient, so we defined a minimum range threshold for splicing. If a guess was known to be in a range smaller than the minimum threshold, we sort the array section instead of splicing it and look up the guesses using a binary search. These optimizations helped minimize the sequential overhead of our method.

C. All-to-All Communication Optimization

The all-to-all communication stage can become the most time consuming part of the algorithm when the number of processors is large. Simply sending out all of the point to point messages at the same time can lead to network overload and contention problems. If we do weak scaling (keeping n/p constant), the amount of data on the network varies in proportion to n, and the number of messages varies in proportion to p^2. Therefore the all-to-all data movement is the most significant scaling bottleneck of Histogram Sort.

We did not utilize any optimized CHARM++ or MPI all-to-all calls [15], [16], because they do not allow for sufficient overlap of communication with computation. Instead, we propose a few simplistic but helpful optimizations to the communication pattern, then focus on preventing the all-to-all from becoming a bottleneck by adapting the algorithm to accomplish significant overlap.

The first basic yet valuable optimization is to have every processor send data in a varying order instead of all cores sending data to processor 0 first and processor $p-1$ last. A simple and effective solution is to have processor p_k send the first message to processor p_{k+1} and wrap around with the last message being sent to processor p_{k-1}. This trivial fix can actually significantly decrease contention on almost any network.

Another straightforward improvement is achieved by staging the communication. Instead of sending all of the messages out immediately, every processor sends out some fraction of its messages in interleaving stages dictated by the amount of data it had received. This approach to collective communication prevents the network from being overloaded, which can cause messages to get delayed, and simplifies the problem of contention on the network. We eventually extended this scheme further by having processors send out one message every time they receive a message. Communication staging improved the all-to-all speed and scaling significantly and completely prevented messages from being delayed. This technique also reduces the memory footprint of the algorithm when considering the use of message buffers.

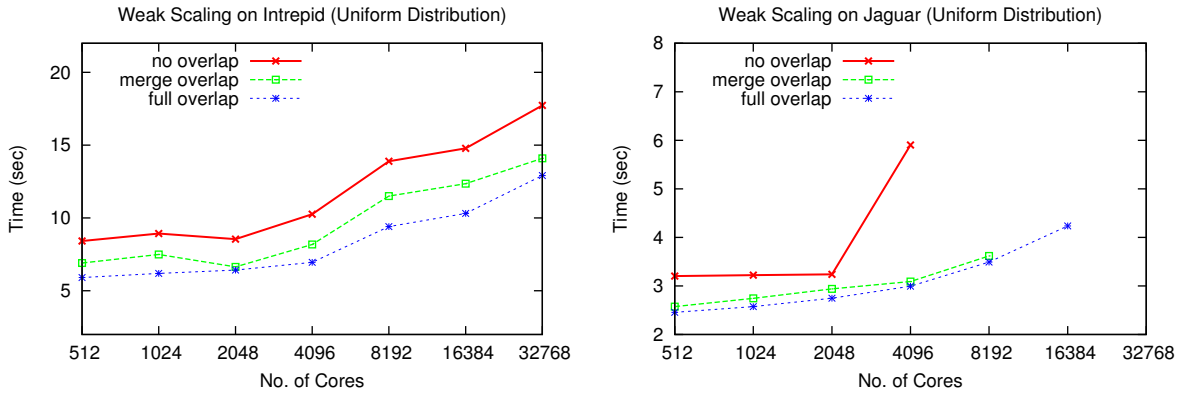

Fig. 2: This figure demonstrates the effect of merge and all-to-all overlap as well as sort and all-to-all overlap. These runs were done on Intrepid and Jaguar with 8 million 64-bit keys per core.

D. Merging and All-to-All Overlap

In the last phase of the algorithm, each processor merges p chunks of sorted data that have been received from all processors. The complexity of merging p sorted arrays of total size n/p is $\Theta(\frac{n}{p} \lg p)$. Since $\frac{n}{p}$ stays constant as we scale up, the $\lg p$ factor dictates the relative growth of the work done in the merging stage. Due to the expanding cost of the all-to-all and merge, we designed our merging algorithm to be capable of exploiting overlap between communication and computation.

There are two basic approaches to merging l arrays into one. One is to do a l-way merge by iterating through every array simultaneously and pushing and popping elements into a heap of size l. Another approach is to merge the arrays in pairs using a merge tree. In a l-way merge there is a $\Theta(\lg l)$ cost for the insertion of an element into the heap, whereas in the tree merge, every element needs to be merged $\Theta(\lg l)$ times. Therefore, both approaches result in the same asymptotic complexity.

A l-way merge is easier to implement and can sometimes outperform a tree merge. Nevertheless, a proper version of a tree merge minimizes any disadvantages of the technique. A tree merge should be implemented using two large arrays (each of size n/p) for the whole of the incoming data in order to avoid unnecessary memory allocations. The data should jump back and forth between the two arrays as it gets merged.

There is an important reason for optimizing and using a tree merge instead of merely utilizing a l-way merge. A l-way merge requires all of the data to be in place to begin, while a tree merge can start as soon as two chunks of data arrive. In fact, the structure of the tree enables us to merge some incoming data upon arrival without any loss of efficiency. In other words, the incoming arrays are being placed at leaf positions of a full binary tree, and are being merged (moved up the tree) with data in every full adjacent node. So for the ith array that arrives, we can do m merges, where $m = max_s[i = 0(mod(2^s))]$, i.e. every other array can be merged immediately at least once, every fourth array can be merged at least twice. This flexibility means that we can merge while the data is arriving, and therefore achieve significant overlap of computation and communication.

This overlap greatly reduces the overhead of the all-to-all communication and improves the scaling of Histogram Sort. The benefit achieved by this merging approach can be observed in Figure 2. The "no overlap" plot is produced by a version of Histogram Sort with no optimizations. The "merge overlap" implementation exploits our all-to-all optimizations and the merging scheme described above. The "merge overlap" version performs noticeably better on both Intrepid and Jaguar. The "full overlap" version is a further-optimized version utilizing the technique described below.

E. Overlapping Sorting with All-to-All

Despite the all-to-all optimizations and merge overlap, we still found that the bottleneck in our performance on a large amount of cores was the mass communication at the end. The algorithm described so far waits for the end of histogramming phase before starting the all-to-all communication. However, we took advantage of our histogramming and sorting overlap technique in a way that allowed us to initiate the data exchange even before local sorting completes.

We found that on a large amount of processors, every histogramming iteration generally determines a significant portion of the total number of desired splitters. By virtue of our splicing technique, we also know that most of the local data has yet to be sorted. Thus, at each histogramming iteration, we also broadcast all determined and bound ranges (range r is bound if splitters s_{k-1} and s_k have been determined), allowing the all-to-all phase

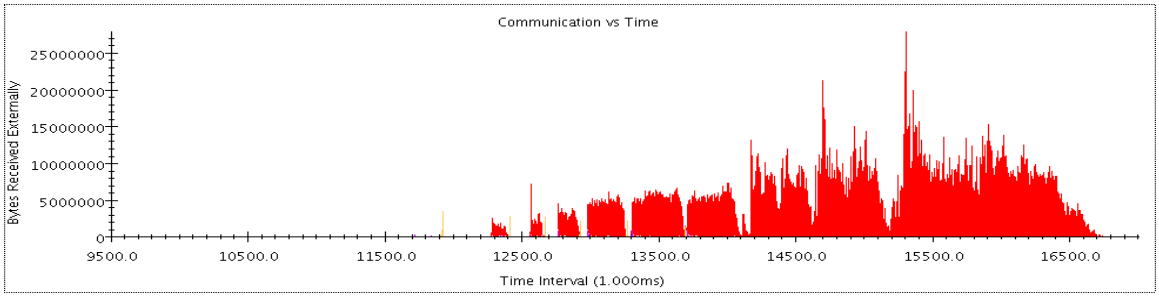

Fig. 3: Communication over Time of Histogram Sort using the Projections tool. This run was done on Intrepid with 8 million 64-bit keys per core on 4096 cores. The plot shows the number of bytes received every millisecond.

to begin early. When a processor receives a bounded range, it sorts the data inside the range and sends it out immediately (the range has already been isolated due to the bounding splitter guesses being used as splicing pivots in a previous iteration).

A fundamental limitation to the benefit of this technique is that all p processors are sending messages to the small subset of processors whose ranges have been determined. This effect can result in contention near the determined processors. Our solution was to only have a fraction of the processors send their data out to any processor owning a newly determined range. This optimization is simply an adaptation of our original communication staging scheme. To select which processors send their data, we define some constant k_{comm}, such that for destination range r, every processor p_i, where $i+r = 0(mod(k_{comm}))$, sends its data to the destination processor p_r. The rest of the processors would send their messages later during the merging stage of the algorithm. We also mirrored our varying-order sends optimization (explained in the previous subsection) by having every processor, p_i, analyze the r_{total} determined ranges starting from destination range $i \ mod \ r_{total}$.

This optimization is especially useful in the case of a relatively uniform distribution, since we are able to send out messages before most of the sorting is complete. It is also equally applicable if the data is non-uniform and several iterations of histogramming are required. In this scenario, we will gradually satisfy all of the splitters, and send out new data at every iteration. Therefore, we are able to execute the sorting stage, the histogramming stage, and the all-to-all concurrently. In Figure 3, we demonstrate the communication spread achieved by overlapping sorting and merging with communication. This plot reflects the total communication on the network over time. We can see that the reception of data is distributed over a long range of time. However, there are spikes and dips in the plots, suggesting that there is still room for improvement.

IV. Performance Results and Analysis

In this section, we will present our implementation groundwork and details. We will also describe the architectures of Jaguar and Intrepid in order to provide a better understanding of our results. Finally, we will present scaling and efficiency data accompanied by analysis and explanations of performance.

A. Implementation Details

We implemented Radix Sort, Sample Sort, and Histogram Sort using CHARM++. This programming model choice allowed us to utilize the portability and flexibility of the object based decomposition technique provided by CHARM++, as well as the multitude of performance and debugging tools provided by the framework [17], [18]. A CHARM++ program specifies computation as a collection of data-driven objects called chares that are organized into indexed collections called chare-arrays. The CHARM++ dynamic runtime system maps chares onto processors, where they execute work in parallel and communicate between each other using messages. CHARM++'s message driven execution and ability to flexibly overlap communication and computation was also useful for our application. For example, we were able to let the runtime system implicitly determine whether a processor can continue sorting or if a message with a new probe has arrived for analysis. We did not utilize over-decomposition [19] in our tests, since it would increase the total communication load. Nevertheless, our code can be efficiently utilized by CHARM++ applications using over-decomposition, such as ChaNGa. The sequential sorting algorithm used in Sample Sort and Histogram Sort was the C++ Standard Library Sort, which is as previously mentioned, Introsort.

Our Radix Sort implementation was consistent with previous work. We decomposed the data and work involved in the algorithm into a parallel chare array, with each chare being responsible for a set of buckets. At the beginning of every iteration, every chare calculates what portion of its data would go to every bucket. A reduction sums up these values and reports them to a single main chare. This main chare determines the

number of buckets to assign to each chare to achieve proper load balancing. After broadcasting these bucket ranges to all of the chare array members, Radix Sort executes an all-to-all data exchange. We applied the communication optimizations discussed above (sending order and staging) to Radix Sort. However, we were unable to modify Radix Sort to achieve any further communication and computation overlap.

We also implemented Sample Sorting by Regular Sampling. The work was decomposed into a chare array with one chare per processor. The samples were collected on each chare and combined using a reduction. The main chare was assigned to analyze the sample then produce and broadcast the splitters. In order to minimize the work done on the main chare, we designed a specialized reduction that merges the samples at every node in the reduction tree. This distributes the merging work and leaves the main chare with $\Theta(sp)$ merging work instead of $\Theta(sp \lg p)$. The same all-to-all optimizations proposed for Histogram Sort were applied to this algorithm as well. Notably, Sample Sort was the easiest to implement of the three parallel sorting algorithms.

For Histogram Sort, the basic decomposition strategy was similarly an array of chares. Every member of the array has responsibility for some amount of the data. A main chare is responsible for the production of probes and analysis of reduction results. This chare can be mapped either onto its own processor or on the same processor as some member of the chare array. The C++ Standard Library Sort function is well optimized and very efficient. By performing splicing, we are essentially replicating the work of this function. Therefore, we tried our best to optimize the efficiency of our code.

We simulate skewed distributions by performing a binary AND between multiple keys to produce every key. This approach is a commonly used test for seeing how well a sorting algorithm handles non-uniform distributions [12]. We will mostly only refer to the case of a single binary AND as a non-uniform distribution. Doing two or more binary ANDs results in very clumped data, which can become almost impossible to deal with as we scale the total number of keys. For example, past a few thousand processors, two AND operations would result in the total number of zeros higher than n/p. For Histogram Sort, instead of doing more ANDs, we will analyze the case of lowering the threshold within which the splitter locations are satisfied.

B. Architecture Information

Intrepid* is an installation of IBM's Blue Gene/P supercomputer at Argonne National Laboratory (ANL) [20]. The installation contains 40 racks of 1024 nodes, with each node containing 4 cores, giving us a total of 163,840 cores. Every core is a PowerPC 450 running at 850 MHz, achieving a peak performance of 3.4 GFLOPs per core. Each node has a total of 2 GB memory shared between the 4 cores with 3 levels of cache. The network topology of Intrepid is a 3D torus. The peak unidirectional bandwidth of a torus link is 425 MB/sec. Therefore, a total bandwidth of 5.1 GB/sec is shared between the 4 cores of each node.

Jaguar†, at Oak Ridge National Laboratory (ORNL), has 7,832 compute nodes in its XT4 partition, also with 4 cores per node, for a total of 31,328 cores. Each node has a quad-core 2.1 GHz Barcelona AMD Opteron processor with 8 GB of memory and a 3 level cache. Every core has a peak performance of 8.4 GFLOPs. Jaguar has a 3-dimensional mesh network. Each link has a unidirectional bandwidth of 3.8 GB/sec, giving a total of 45.6 GB/sec per node. However, the total node bandwidth may be bound by the HyperTransport link throughput, which is 6.4 GB/sec.

C. Analysis

Figure 4 plots the performance of Radix, Regular Sample, and Histogram Sorts as well as the ideal scaling. We derived the ideal plot by doing a sequential run on Intrepid which sorted 16 million keys using the STL library sort. We then scaled this run using the relation, $running\ time \sim \frac{n \lg n}{p}$. Radix Sort has a large overhead over Histogram Sort. This overhead is likely due to the cache inefficient methods of Radix Sort, as well as the number of times the algorithm looks at each key. Note that we are using 64-bit keys, rather than 32-bit keys, which has a significant effect on the performance of Radix Sort since it requires double the number of all-to-all stages. The scaling of Radix Sort also gets bad at 4096 processors, probably due to the multiple communication stages and lack of overlap between communication and computation. Sample Sorting by Regular Sampling seems to be slightly slower than Histogram Sort but begins deteriorating in performance at 4096 processors once the combined sample gets bigger. At 8192 processors, the combined sample cannot fit into one processor's memory.

On the other hand, Histogram Sort is fairly close to ideal and does not begin to deteriorate in scaling until 8192 cores. Even past this point the deterioration is gradual and slow all the way up to 32 thousand processors. A better look at the speedup of this algorithm can be seen in Figure 5. On Intrepid, we achieve over 80 percent efficiency until 512 cores, where it begins to slowly deteriorate. At 32 thousand cores, we are barely under 50 percent efficiency, which means we achieve a

*http://www.alcf.anl.gov/support/usingALCF/usingsystem.php

†http://nccs.gov/computing-resources/jaguar/

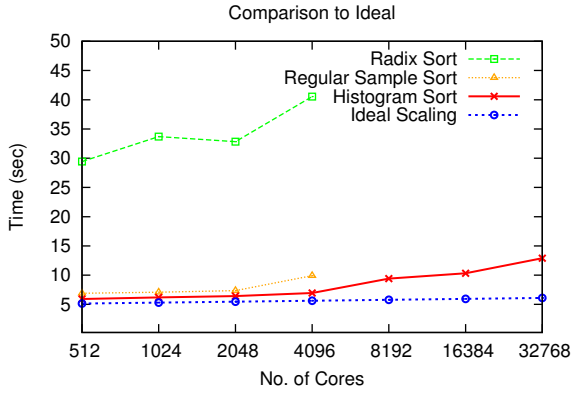

Fig. 4: This figure compares the performance of Histogram Sort, Sample Sort, and Radix Sort (16-bit radix) to the ideal performance. These runs were done on Intrepid with 8 million 64-bit keys per core.

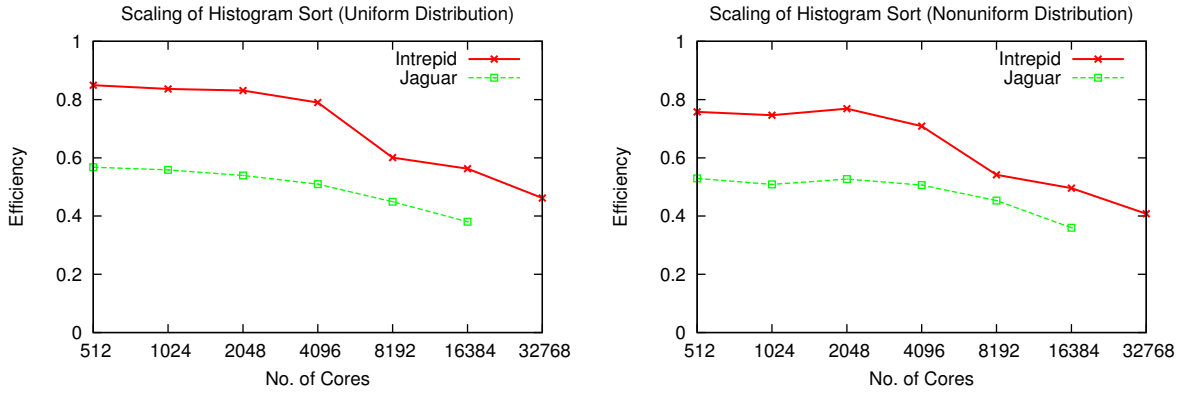

Fig. 5: This figure shows the proportion of maximum speedup achieved by Histogram Sort given uniform and non-uniform distributions. These runs were done on Intrepid and Jaguar with 8 million 64-bit keys per core.

speedup of about 15,000. On Jaguar, the efficiency is a bit worse. However, the execution time stays low and scales well. Despite the lower speedup, Histogram Sort actually performs almost three times faster on Jaguar than on Intrepid. Both of these differences are probably due to the fact that Jaguar has significantly faster cores and finishes the sequential sorting much quicker.

Looking at the non-uniform distribution speedup graph and the left graph of Figure 6, we can first observe that the extra cost of finding splitters for a skewed distribution is fairly small and the scaling is roughly the same for either type of distribution. In fact, the loss is virtually nothing for Jaguar, which is able to handle the reductions very quickly and successfully exploit overlap. On Intrepid, there is a slowly expanding overhead, but the speedup curve has roughly the same shape. These results indicate that the iterative histogramming strategy and overlap achieve their goal with a cost that is marginal with respect to the execution time. Decreasing the splitter threshold increases the number of iterations necessary to achieve sufficiently good splitting. As shown by the right graph in Figure 6, the extra iterations are very cheap at 4k processors and can actually be beneficial due to improvement to the load balance of the merging stage. On the other hand, at 16k processors every iteration adds a bit of overhead (halving the threshold usually adds about one iteration).

Overall, Histogram Sort achieves good efficiency and scaling up to about 4096 processors, then begins to deteriorate in efficiency compared with the ideal. However, we maintain good performance to 32k processors in comparison to Radix Sort, which seems to suffocate in efficiency at around 4096 processors, and the original version of Histogram Sort, which shows a continuously growing performance deficit (refer to Figure 2). Our optimizations improve execution time for any processor range and slow down the deterioration in efficiency. Without all-to-all optimizations and overlap of the merging work with communication, this type of scaling could not be achieved. An instructive example of this is reflected in Figure 2, as Jaguar stops scaling at 4096 cores without optimizations. This sharp slope is probably due to messages getting delayed and arriving very late. Our strategies for communication optimization and overlap of communication with computation are able to distribute communication and prevent this type of

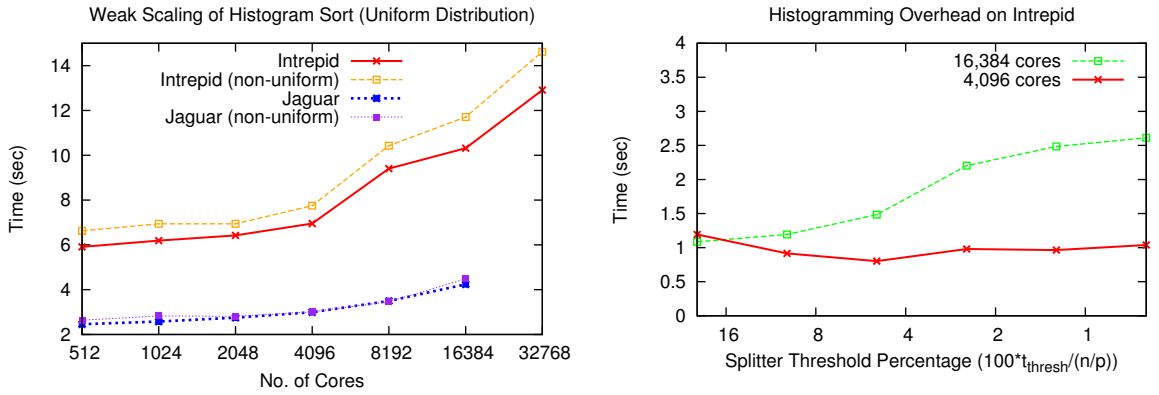

Fig. 6: The figure on the left shows the weak scaling of Histogram Sort given uniform and non-uniform distributions. The figure on the right displays the histogramming overhead of decreasing the splitter threshold. These runs were done on Intrepid and Jaguar with 8 million 64-bit keys per core.

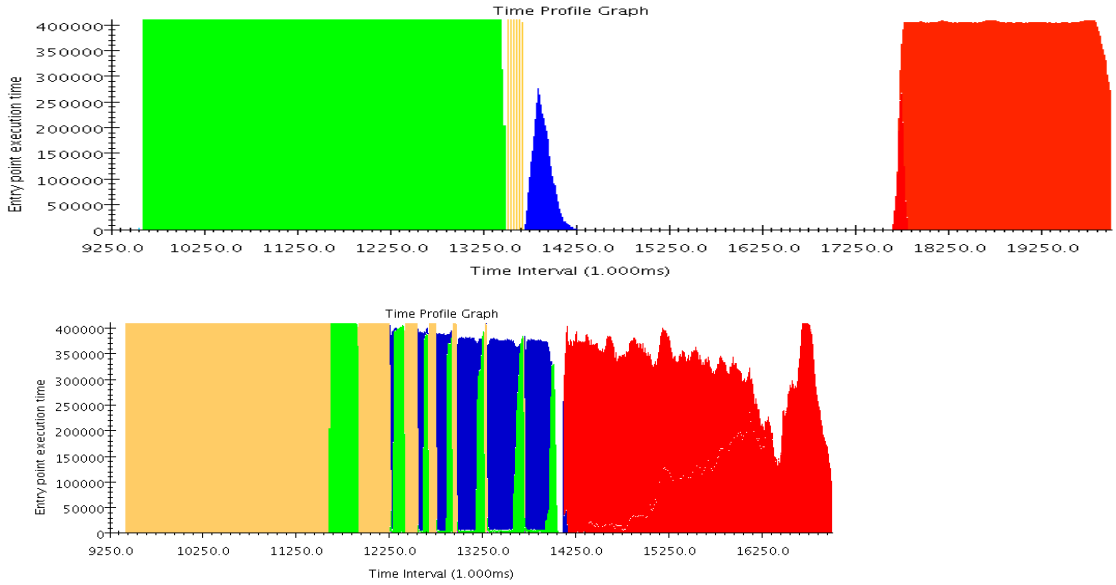

Fig. 7: Time Profile of Histogram Sort using the Projections tool. The first image demonstrates the utilization graph of the non-optimized Histogram Sort. The second image shows the utilization graph of the optimized Histogram Sort. The colors are the same as those in Figure 1. These runs were done on Intrepid with 8 million 64-bit keys per core on 4096 cores.

delay, as well as reduce the critical path of the algorithm. A good visualization of these effects can be seen in the time profiles shown in Figure 7. The large amount of idle time (from approximately 14.25 to 17.5 seconds) in the non-optimized version is due to the blocking all-to-all communication stage. Our improved version avoids this heavy cost by maximizing overlap.

The long first orange block in the time profile of the optimized version is actually performing the first splicing iterations. At first sight, these might seem to be taking way too long. However, if we are trying to locate the first p splitter guesses, the fraction of the total sorting work done finding them is approximately $\frac{\frac{n}{p}\lg p}{\frac{n}{p}\lg n/p}$, or $\frac{\lg p}{\lg n/p}$. Thus for $p = 4096 = 2^{12}$ and $n/p = 2^{23}$, we do roughly one half of the total sorting work just finding the first $p-1$ splitter guesses. The amount of time spent sending might also look to be taking much longer in the optimized version, but it actually includes some of the sorting. Due to our expedited communication strategy we often still need to sort the data before sending it.

The time profiles provide us with information on how close we get to peak performance in any given section of the algorithm. The optimized version of Histogram Sort is able to avoid being idle and stays at a high percentage of peak during the histogramming stage and the merging stage. Evidently, the non-optimized version wastes a significant amount of time waiting for the blocking communication. This difference is amplified as we use more processors since the histogramming and all-to-all stages become more expensive.

V. Conclusion

Our extensions to Histogram Sort have demonstrated an improvement to the previous parallel sorting strategies and shown good scaling results utilizing significant portions of large modern supercomputers such as Blue Gene/P and XT4. We employ methods which are all generalizable to other architectures and adaptable to different histogramming and communication approaches. Therefore, we think our methods can be realized as improvements to previously established algorithms and parallel programs.

The most practical benefit of our work is a potential improvement for parallel scientific applications doing parallel sorting. We have already used our optimizations in one such application, a parallel cosmology simulation code, ChaNGa. The histogramming and merging strategies showed good performance and scaling in ChaNGa and can likely improve the speed of other applications.

The performance benefits achieved by our optimizations prove the importance of smart communication strategies and overlap of communication with computation to scaling on modern parallel machines. Our study also demonstrates that scaling and parallel efficiency can be improved by making concessions and adaptations to the sequential algorithm in order to obtain better flow and distribution of communication.

Future work might attempt to create an API for parallel sorting so that our algorithm or other strategies can be easily interchanged and integrated into parallel code. This task is non-trivial due to the unknown distribution of data and the variety of applications of parallel sorting.

Histogram Sort might also be improved by adopting a scheme introduced in a sequential sorting algorithm, Super Scalar Sample Sort [21]. This sorting algorithm is able to reduce the conditional branching cost of Quicksort by recursively collecting data samples and moving the data into buckets defined by those samples. The algorithm makes interesting use of predicated instructions to quickly search through heaps. On certain processors, Super Scalar Sample Sort outperforms the C++ Standard Library Sort. Histogram Sort might benefit from using this algorithm for local sorting by utilizing the probe to define the buckets instead of collecting a sample. The current technique of using elements of the probe as Quicksort pivots yields a sequential overhead, while this scheme might even yield a sequential speedup.

Attempts to further scale Histogram Sort to higher numbers of cores will need to take more careful consideration of the performance of the histogramming stage. We have observed rapid growth in histogramming iteration time at the 16k-32k processor range (an iteration can take up to half a second on 32k cores). One way to tackle this problem would be to utilize early knowledge of defined ranges to split up the histogramming into multiple sections. Once we determine a few middle splitters, we can parallelize the creation and analysis of probes efficiently by having a probe only target a certain range of splitters. This approach should slice up the bandwidth of the reductions at the cost of increasing latency and contention on the network. More importantly, it would significantly speedup the probe analysis time by parallelizing the process.

As parallel machines continue to grow, we have to reconsider widely accepted parallel algorithms and paradigms since the challenges they are designed to solve are quickly changing. The parallel sorting research done 10 years ago is no longer good enough for modern supercomputers, and our study will likely not be sufficient for optimal scaling on future parallel machines. However, we have tried to lay a framework for parallel sorting algorithms that is highly scalable on current supercomputers and has no inherent theoretical flaw preventing it from scaling further.

Acknowledgements

This work was supported in part by DOE Grant DE-FG05-08OR23332 through ORNL LCF. This research used the Blue Gene/P at Argonne National Laboratory, which is supported by DOE under contract DE-AC02-06CH11357. The research also used Jaguar at Oak Ridge National Laboratory, which is supported by the DOE under contract DE-AC05-00OR22725. Accounts on Jaguar were made available via the Performance Evaluation and Analysis Consortium End Station, a DOE INCITE project.

References

[1] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Massively Parallel Cosmological Simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[2] J. E. Barnes and P. Hut. A hierarchical O(NlogN) force calculation algorithm. *Nature*, 324, 1986.

[3] L. V. Kale and Sanjeev Krishnan. A comparison based parallel sorting algorithm. In *Proceedings of the 22nd International Conference on Parallel Processing*, pages 196–200, St. Charles, IL, August 1993.

[4] K. Batcher. Sorting Networks and Their Application. volume 32, pages 307–314, 1968.

[5] Gianfranco Bilardi and Alexandru Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. Technical report, Ithaca, NY, USA, 1986.

[6] G. Blelloch et al. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proc. Symposium on Parallel Algorithms and Architectures*, July 1991.

[7] W.D. Fraser and A.C. McKellar. Samplesort : A sampling approach to minimal storage tree sorting. *Journal of the Association for Computing Machinery*, 17(3), July 1970.

[8] J.S Huang and Y.C Chow. Parallel sorting and data partitioning by sampling. In *Proc. Seventh International Computer Software and Applications Conference*, November 1983.

[9] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.

[10] Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong, and Hanmao Shi. On the versatility of parallel sorting by regular sampling. *Parallel Comput.*, 19(10):1079–1103, 1993.

[11] John H. Reif and Leslie G. Valiant. A logarithmic time sort for linear size networks. *J. ACM*, 34(1):60–76, 1987.

[12] Kurt Thearling and Stephen Smith. An improved supercomputer sorting benchmark. In *Proc. Supercomputing*, November 1992.

[13] David R. Cheng, Alan Edelman, John R. Gilbert, and Viral Shah. A novel parallel sorting algorithm for contemporary architectures. *Submitted to ALENEX06*, 2006.

[14] David R. Musser. Introspective Sorting and Selection Algorithms. *Sortware: Practice and Experience*, 27(8):983–993, January 1997.

[15] L. V. Kale, Sameer Kumar, and Krishnan Vardarajan. A framework for collective personalized communication, communicated to ipdps 2003. Technical Report 02-10, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2002.

[16] Rajeev Thakur and William D. Gropp. Improving the Performance of Collective Operations in MPICH. *Lecture Notes in Computer Science*, 2840:257–267, October 2003.

[17] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, 1993.

[18] Chee Wai Lee, Terry L. Wilmarth, and Laxmikant V. Kalé. Performance visualization and analysis of parallel discrete event simulations with projections. Technical Report 05-19, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, November 2005. Submitted for publication.

[19] Laxmikant V. Kale. Some Essential Techniques for Developing Efficient Peta scale Applications . July 2008.

[20] IBM Blue Gene Team. Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development*, 52(1/2), 2008.

[21] Peter Sanders and Sebastian Winkel. *Algorithms ESA - 2004*, chapter Super Scalar Sample Sort, pages 784–796. Springer Berlin / Heidelbeg, 2004.