Parallel Sorting Pattern

Vivek Kale
University of Illinois, Urbana, IL, USA
vivek@illinois.edu

Edgar Solomonik
University of California, Berkeley, CA, USA
solomonik@berkeley.edu

1. PROBLEM

A large number of parallel applications contain a computationally intensive phase in which a large list of elements must be ordered based on some common attribute of the elements. How do we sort a sequence of elements on multiple processing units so as to minimize redistribution of keys while allowing processing units to do independent sorting work?

2. CONTEXT

Sorting is the process of reordering a sequence taken as input and producing one that is ordered according to an attribute. Parallel sorting is the process of using multiple processing units to collectively sort an unordered sequence. The unsorted sequence is composed of disjoint sub-sequences, each of which is associated with a unique processing unit. Parallel sorting produces a fully sorted sequence composed of ordered sub-sequences, each of which is associated with a unique processing unit. The produced sequences are typically ordered according to the given processor ordering and are of roughly equal length.

It is important to realize that *parallelization* of sorting algorithms (particularly under the distributed memory model which we focus on) is a fundamentally different problem from *acceleration* or *hardware performance tuning* of a sorting algorithm. Indeed, such acceleration and tuning is often implemented using techniques similar to parallel sorting algorithms implemented within a distributed memory model [1] [2]. However, the semantics and goals of such acceleration and tuning of sorting is often different than that of parallel sorting under the distributed memory model and deserves separate analysis that will not be provided here. In particular, we focus on sorting a sequence composed of a set of sequences, each associated with a logical memory.

Parallel sorting under the distributed memory model is a fundamentally different problem from sorting under a sequential or shared memory programming model. Sequential sorting takes an unsorted sequence as input and outputs a single sorted sequence. Sequential sorting can be accelerated using parallel processing provided by shared memory multicore machines or accelerators such as GPUs. Such parallel acceleration is often performed using algorithms similar to those used in distributed memory sorting. Nevertheless, the semantics of such parallel acceleration are different than those of the distributed memory model and deserve separate analysis that will not be provided here.

In designing parallel sorting algorithms, the fundamental issue is to collectively sort data owned by individual processes in such a way that it utilizes all processing units doing sorting work, while also minimizing the costs of redistribution of keys across processors.

While sequential sorting libraries such as STL sort [3] often eliminate the need for the programmer to think about implementation details of a sorting algorithm, justifying the use of a standard library for a parallel sorting algorithm is much more difficult. In order to implement a sorting algorithm within a parallel programming model, there are a wide array of sorting solutions to consider. Each of these solutions cater to a parallel application and/or a particular machine architecture/platform. In general, implementing a parallel sorting kernel requires the programmer to consider many different design decisions with careful consideration of the application characteristics.

Parallel sorting algorithms are important computational kernels used in support of both high performance scientific applications and industrial applications. In parallel scientific benchmarks involving cosmology simulations(e.g. Barnes-Hut), the oct-tree building operations often require sorting the keys before the actual tree building phase [4]. Parallel sorting algorithms may also need to be implemented for travel websites such as expedia.com or travelocity.com where many different flights must be collected from each air carrier's database, and the cheapest flights across all air carriers must be displayed to the user.

In many applications, certain initial distributions recur in the application. Some common initial distributions are uniform, mostly sorted, completely random, reverse order. Just as with sequential sorting algorithms, parallel sorting algorithms are sensitive to the initial distribution of keys.

However, there are many more ways to characterize the

ParaPLoP 2010

initial distributions when considering parallel sorting algorithm. A parallel sorting algorithm must consider not only the sequence of keys, but should also consider the local distributions of keys on each processor. How does a parallel sorting algorithm behave with a mostly ordered sequence? What if almost all values of keys are zero? What if keys are locally sorted within each processor, but not globally sorted? How well does a parallel sorting algorithm handle uniform distributions?

In implementing a sequential sorting algorithm, the cost of rearranging keys is proportional the cost of reading or writing to memory or disk. In a parallel sorting algorithm implementation, rearranging keys requires communication over an interconnect network, and the cost of this data movement is often non-trivial: there is a high probability that keys are moved from one processor to another (namely, $\frac{p-1}{p}$, where p is the number of processors).

Another important consideration of initial distributions is the size of keys(e.g. is the representation 32-bit numbers or 64-bit numbers?). With increasing key sizes, the cost of moving a key from one processor to another increases. Thus, one must be very careful to implement a parallel sorting algorithm in a way that maintains locality and minimizes data movement. Also, the ratio of keys to processors (i.e. $\frac{n}{p}$) becomes very important in parallel sorting.

When $\frac{n}{p}$ is large, minimizing movement of keys over the interconnect becomes a priority. When $\frac{n}{p}$ is small, the communication latency becomes important. If $\frac{n}{p} \approx 1$, the cost of communication for redistribution of keys completely dominates any local sorting computation. If $\frac{n}{p}$ approaches infinity, the cost of the sorting computation will completely dominate the performance. Indeed, if $\frac{n}{p} \leq 1$, it is usually a good idea to sort on a single processor or a subset of the p processors.

In general, as more parallel applications are being developed today, more innovative parallel sorting algorithms need to be designed to support these applications. As sorting algorithms are sensitive to initial distributions, understanding the context and the needs of the end application is important in designing and implementing parallel sorting algorithms. The proper assessment of application knowledge often can suggest which initial distributions are likely to occur, allowing the programmer to implement a sorting algorithm that works effectively for that application.

While some of the parallel formulations of sorting algorithms may be relatively recent, there has been extensive knowledge accumulated about parallel sorting in the last 40 years, and a variety of literature can be found on the description of different parallel sorting algorithms in [5] and [6].

In this paper, our goal is not to simply provide a survey of the plethora of currently used parallel sorting algorithms. Rather, we provide a documentation of key ingredients required for effective parallel sorting by formulating parallel sorting as a parallel programming pattern.

By discussing representative parallel sorting solutions, we aim to guide programmers to realize what design considerations need to be made for parallel sorting algorithm, so that a programmer can customize a sorting algorithm to work for their specific application.

The following sections discuss general guidelines for implementing parallel sorting algorithms, document fundamental and widely used parallel sorting algorithms, and make note of forces one must consider when implementing parallel sorting algorithms.

3. FORCES

3.1 Universal Forces

3.1.1 Ability to exploit partially sorted initial distributions

Since sorting is often done periodically on the same sequence, in such cases, the ordering of the sequence is usually only perturbed. Some parallel sorting algorithms can take advantage of this scenario and perform less work or less data movement.

This force is particularly important for distributed memory sorting algorithms, since a partially sorted sequence usually implies that much less data needs to be moved between processors. If the pertrubation of all elements from their sorted positions is small enough, most processors can keep most of their initial data and data movement may be restricted to neighboring processors.

Parallel sorting algorithms exploit these advantages to varying degrees. At application level, the choice between these sorting algorithms should first take into consideration the relevance of this force. Then, if neccessary, the decision should account for the corresponding force resolutions of the algorithmic options.

3.1.2 Data movement

The movement of data from one processor to another must be minimal throughout the execution of the algorithm. The cost of migrating data between processors on a distributed memory system can dominate the overall execution cost and is a fundamental scalability bottleneck.

For a distribution which has no locality (every element belongs on a different processor rather than its staring location), the minimal bound on data movement in n, since every element must move at least once. In fact, this minimal bound is asymptotically correct for the majority of distributions, unless the keys are mostly sorted initially.

Some parallel sorting algorithms achieve the minimal communication bound. However, these algorithms tend to have a larger latency cost and some struggle to maintain load balance efficiently. Most of these algorithms are *splitter-based*. This term as well as a few of such algorithms are discussed below.

3.1.3 Load Balance

The purpose of sorting in any context is usually to organize data in a way that simplifies or accelerates future operations. However, parallel sorting must, additionally, ensure *load balance* in its organization of the data. Virtually all

parallel applications require that data be distributed evenly throughout processors. The importance of load balance is immense, since application execution time is typically bound by the local execution time of the most overloaded processor.

Any modern parallel sorting algorithm has a mechanism for maintaining load balance. However, these mechanisms typically achieve varying levels of load balance at an inversly related algorithm complexity cost. Therefore, much of the insight and merit of a parallel sorting algorithm is the quality of its load balancing technique.

We can make some assertions on the quality of load balance in a parallel sorting algorithm, such as its parallel scalability and asymptotic complexity. However, the actual performance of the given load balancing technique can be tied closely with the initial distribution and therefore the application semantics.

3.2 Implementation Forces

3.2.1 Communication latency

Latency is the average time it takes to send a message from one processor to another. Optimizing for low latencies may not be worthwhile for machines with a small number of processors. However, latency becomes very important for large-scale distributed machines.

Especially if $\frac{n}{p}$ is small, data movement latency can take a large toll on the scalability and performance of a parallel sorting algorithm. Given an initial distribution where every processor contains an key belong to a different processor, the minimal latency is $p \log p$ messages. This minima is asymptotically correct for most randomized distributions.

Unfortunately, the minimal latency bound can be achieved only via combining messages. This condition means that both minimal latency and minimal data movement cannot be acheived, since combining messages forces a piece of data to move multiple times.

Parallel sorting algorithms also incur a latency cost from their load balancing techniques and orchestration of data movement. Taken all together, these latency costs can become a bottleneck for parallel sorting on smaller data sets on a large amount of processors.

3.2.2 Supplementary communication bandwidth

Bandwidth is the aggregate data rate at which processors send and receive keys. This refers to the bandwidth *acheived* by the algorithm, rather than the bandwidth *available* in the interconnect network of the machine. It is also important to realize that by aggregate, we mean that the bandwidth is summed over the entire machine. We are not dealing with per-processor bandwidth.

The communication bandwidth is practically always dominated by the data movement. Further, the minimal communication bandwidth bound for a randomized distribution is $Theta(n)$. However, all parallel sorting algorithms typically do some additional communication in order to achieve load balance or orchestrate the data movement. This supplementary communication bandwdith of a parallel sorting algorithm is often becomes main parallel scalability bottleneck when sorting smaller data sets on a lare number of processors.

3.2.3 Overlapping communication with computation

In any parallel application, if there is a significant cost in communication as well as sequential computation, the execution time can often be significantly reduced (up to 2x) if overlap is exploited between the two tasks. Typically, a parallel sorting tasks needs to perform a significant amount of sequential sorting work as well as data movement. Thus, overlap *can* give a significant performance boost.

Achieving communication and computational overlap often requires substantial programming and optimization work. Moreover, for some parallel sorting algorithms it can be more difficult or impossible to exploit a significant amount of overlap. Other algorithms, have a more relaxed critical path and sorting work can be pipelined in ways that achieve significant overlap between communication and sequential computation.

4. SOLUTION

4.1 General Solution

4.1.1 General terminology

The following terms will be used repeatedly throughout the solution section.

- n - total number of elements to be sorted.
- p - number of processing units
- $\Pi_{[1,p]}$ - unsorted initial sequences. Processing unit k posseses Π_k for $k \in [1, p]$.
- $len_{[1,p]}$ - length of initial sequences. Π_k has length k for $k \in [1, p]$.
- $\Xi_{[1,p]}$ - sorted final sequences. Processing unit k posseses Ξ_k for $k \in [1, p]$.
- $flen_{[1,p]}$ - length of final sequences. Ξ_k has length $flen_k$ for $k \in [1, p]$.
- $threshold$ - the maximum number of extra keys each processor can end up with. More precisely, for all $k \in [1, p]$, $flen_k \leq \frac{n}{p} + threshold$.

4.1.2 Splitting of data

A fundamental problem of parallel sorting is ensuring that each Ξ_k has a continuous portion of the entire key set. This requirement is necessary to achieve globally sorted order over all processors. The requirement could be rephrased to say that for $k \in [2, p-1]$, Ξ_k has all existing keys in the range $[Sptr[k-1], Sptr[k]]$, Ξ_1 has all keys smaller than $Sptr[1]$ and, Ξ_p has all keys larger than $Sptr[p-1]$. $Sptr$ is commonly referred to as a splitting vector and has length $p-1$.

Most modern parallel sorting algorithms find $Sptr$ explicitly in order to determine what processor each key belongs on. The difficulty lies in that $Sptr$ directly dictates the load balance of the final sequence sizes. As previously mentioned,

$flen_k$ should be smaller than $\frac{n}{p} + threshold$ for all $k \in [1, p]$. It is therefore necessary to find a splitting vector, $Sptr$, such that for each of p key ranges generated by elements $Sptr$, the number of existing keys in the range is smaller than $\frac{n}{p} + threshold$.

At a high level, we can gain the insight that a splitting vector needs to properly relate the distribution of keys within the key range with the corresponding key values.

There are three commonly used methods for determining the splitting vector, $Sptr$:

1. Pre-emptive: Use application-level knowledge to establish $Sptr$ directly. Or, simply assume $Sptr$ to be uniform.
2. Sample [7] : draw a sample from the global key set and select $Sptr$ from the sample.
3. Histogram [8]: make a naive guess at $Sptr$, then iteratively adjust it by computing how many keys belong to each range.

Many parallel sorting algorithms use modified versions of these techniques but most use at least one at a high level. Notably, some algorithms do not determine $Sptr$ all at once but rather in multiple iterations or levels of recursion.

4.1.3 Control flow of parallel sorting Algorithms:

From a communication based perspective a rough control flow can be generally defined for the parallel sorting pattern. We can single out three computatonally heavy sequential functions often performed by parallel sorting algorithms:

1. Local sorting: the local keys on each processor are often completely sorted initially or sorted in chunks at some point of execution.
2. Bucketing: it is often necessary to place keys into buckets in order to send them or to compute histograms.
3. Merging: it is often the case that keys are obtained in sorted sub-sequences and need to be merged into a complete sequence.

In order to make a generalization about the control flow of sorting algorithms, we crudely designate the above functions as well as any other miscellaneous sequential functions as *local work*. The majority of parallel sorting algorithms can now be generalized as follows:

1. Do local work.
2. Collect distribution-relevant information from all processors.
3. On a single processor, infer a splitting of keys from the collected information.
4. Broadcast the splitting elements.
5. Do local work.
6. Move data according to the splitting elements.
7. Do local work.
8. If the splitting was incomplete (not all of $Sptr$ defined) recurse by goin back to step 1.

This analysis gives us a few insights towards communication efficiency of parallel sortng algorithms. First, there are two major communication functions, figuring out a global splitting vector and transposing the data to the proper processors. The second insight is that most algorithms have multiple stages of local work and it can be very beneficial to try to overlap this local work with the communication. Finally, if overlap can be achieved between local work and the third step (sequential splitter determination), a processor can be reserved for the splitting work, shortening the critical path.

The cost of the communication a parallel sorting algorithm needs (to infer the splitting and move the data) and the cost of local work that can be overlapped with, gives us a very good idea of the comparative parallel scalability of parallel sorting algorithms.

4.2 Parallel Quicksort

Quicksort applies a divide-and-conquer approach by recursively partitioning the sequence using a pivot element. A very basic sequential quicksort algorithm is described by Leiserson et al [3]. The divide-and-conquer algorithmic structure is naturally extensible to a parallel formulation.

Parallelization is typically achieved by using the pivots to recursively partition the set of interacting processors. Load balance is maintained by the semantics of pivot selection.

4.2.1 Terminology

- $Procs_{[1,p]}$ - a set of processors.
- *pivot* - a key used to subdivide sequences of elements.
- *num_procs_s* - number of processors to receive elements smaller than *pivot*.
- $Procs\_s_{[1,num\_procs\_s]}$ - a set of processors designated to receive elements smaller than *pivot*.
- *num_procs_l* - number of processors to receive elements larger than *pivot*.
- $Procs\_l_{[1,num\_procs\_l]}$ - a set of processors designated to receive elements larger than *pivot*.
- $Small_{[1,p]}$ - the sequence of elements that are smaller than *pivot* on processor k.
- $len\_s_{[1,p]}$ - lengths of $Small$ sequences. $Small_k$ has length len_s_k for any key *pivot*.
- $prefix\_s_{[1,p]}$ - the prefix summation of len_s.
- $average\_s$ - $\frac{prefix\_s_p}{num\_procs\_s}$.

- $Large_{[1,p]}$ - the sequence of elements that are larger or equal to *pivot* on processor k.
- $prefix\_l_{[1,p]}$ - the prefix summation of len_l.
- $average\_l$ - $\frac{prefix\_l_p}{num\_procs\_l}$.
- $len\_l_{[1,p]}$ - lengths of $Large$ sequences. $Lare_k$ has length len_l_k for any key *pivot*.
- *master* - a processor designated to broadcast, collect, and analyze global information.

4.2.2 Example Parallel Quicksort Algorithm

To illustrate parallel quicksort, we begin with an example algorithm which we refer to as GramaQuickSort. See [6] for more details. The algorithm takes as input the set of processors, $Procs$ and the list of keys Π that it operates on. The algorithm outputs a globally sorted sequence of keys Ξ.

1. If $p = 1$(the group consists of 1 processor), sort local data on the processor in this set and STOP. Merge the final subsequences. Otherwise, continue to step 2.
2. The *master* processor of set $Procs$ broadcasts a key, *pivot*, to all processors in its set.
3. Each processor k in set $Procs$ partitions its keys, with respect to the pivot, into two sets $Small_k$ and $Large_k$ of length len_s_k and len_l_k, respectively.
4. len_s is summed over all p processors in the set of processors $Procs$. len_l is summed over all p processors in the set of processors $Procs$. The sizes of sets len_s and len_l are used in a parallel prefix operation to determine the target processors for all keys in such a way that each processor has an equal number of keys.
5. The master processor of set $Procs$ partitions the set of processors into two subsets $Procs\_s$ and $Procs\_l$ of sizes num_procs_s and num_procs_l, respectively. Specifically, the master processor decides a subset of processors $Procs\_s$ which should be given the smaller keys ($Small\_s$) and the subset of processors $Procs\_l$ that should be given larger keys ($Large\_l$).
6. At this point, processor $p-1$ knows the total sums, $prefix\_s_p$ and $prefix\_l_p$. Processor $p-1$ broadcasts the average number of keys these two sets of processors should receive ($average\_s = \frac{prefix\_s_p}{num\_procs\_s}$ and $average\_l\frac{prefix\_l_p}{num\_procs\_s}$)
7. Processor k's sets of keys $Small_k$ and $Large_k$ are redistributed *within this set of processors* P as follows.

   Keys $Small_k$ will be sent to processor $\lfloor prefix\_s_{k-1}/average\_s \rfloor$ through processor $\lceil prefix\_s_k/average_s \rceil$. Keys $Large_k$ should go to processor $\lfloor prefix\_l_{k-1}/average\_l \rfloor$ through processor $\lceil prefix\_l_k/average_l \rceil$. Recall that the parallel prefix operation in step 4 determines exactly which processor a key in $Small_k$ would go to.
8. With this, the processors in $Procs_s$ have smaller keys and processors in $Procs_l$ have larger keys. *Recurse* GramaQuickSort($Procs_s$) in parallel with GramaQuicksort($Proc$

4.2.3 Resolution of Forces

1. **Load Balance**

   Resolution: Very Good (4/5)

   Explanation: This parallel quicksort algorithm is well load balanced because the prefix sum operation ensures that there are $\frac{n}{p}$ keys on each processor in every phase. Randomization and sampling techniques can further improve load balance.

2. **Data Movement**

   Resolution: Very Good (4/5)

   Order of Growth: $\Theta(nlogp)$

   Explanation: Data movement is relatively high because for each of the $log(p)$ phases, there is approximately a 50% probability that each key will need to be moved to another processor. Since the expected number of keys moved within each phase is $\frac{n}{2}$, and there are $log(p)$ phases, data movement is on the order of $\Theta(nlogp)$.

3. **Communication Latency**

   Resolution: Excellent (5/5)

   Order of Growth: $\Theta(plogp)$

   Explanation: There are two primary factors in parallel quicksort that contribute to latency. First, each phase requires the master processor within a processor group to broadcast pivot to the other processors within that group. Accumulated over $log(p)$ phases, broadcasting pivots results in communication latency on the order of $\Theta(plogp)$. Second, the parallel prefix sum within each recursive step ensures that the set of keys $Small_k$ and $Large_k$ on processor k are typically sent to one or two other processors. The number of messages sent and received by each of the p processors in each of the $log(p)$ phases is small (usually no more than 4). The total number of messages exchanged between processors is on the order of $\Theta(plogp + plogp) = \Theta(plogp)$. As message latency is independent of the number of keys n and since $n >= p$ in most scenarios, communication latency of parallel quicksort is better than many other parallel sorting algorithms.

4. **Supplementary Communication Bandwidth**

   Resolution: Very Good (4/5)

   Order of Growth: $\Theta(plogp)$

   Explanation: In each recursive step, the master processor of each group must broadcast one pivot to all other processors within its group in order to determine how keys are redistributed across processors. In the

first step there are 2 pivots broadcasted within each of the two processor groups. In each subsequent step, the number of pivots broadcasted doubles, since the number of groups doubles. There are $\Theta(logp)$ recursive steps. This results in additional bandwidth that is on the order $\Theta(plogp)$.

5. **Ability to Exploit Initial Distribution**

   Resolution: Very Good (4/5)

   Explanation: If the initial distribution is nearly sorted, then the processors will not need to redistribute many keys since the smallest values stay on lower-numbered processors while largest values stay on higher-numbered processors. Regardless of the method of choosing the pivot, parallel quicksort always will be able to exploit the initial distribution; the more sorted the keys are, the less data that gets moved.

6. **Overlapping Communication with Computation**

   Resolution: Moderate (3/5)

   Explanation: Parallel Quicksort can exploit overlap between data movement and computation by merging data chunks as they arrive. However, every step each processor only receives a few chunks so this flexibility is rather limited.

   If properly implemented, Parallel Quicksort can also achieve some overlap with local work and pivot application and prefix operation. However, this communication is rather insignificant, so the overlap is unlikely to greatly improve performance.

4.2.4 Discussion

It is important to realize that *no system-wide barrier synchronization is needed in all recursive steps.* Once the subsets are partitioned, each group of processors proceeds independently. Within each group, a group-specific reduction and parallel prefix operation is needed. This locally synchronizes only the processors within that group.

In parallel quicksort, all the keys will move out of their home processors during the execution of the algorithm. However, communication latency of messages in a parallel quicksort algorithm increases only with increasing number of processors. Thus, parallel quicksort suffers less communication latency overhead than the all-to-all personalized communication present in parallel radix sort or sample sort.

4.2.5 Other Parallel QuickSort Algorithm Variants

1. Quinn's QuickSort [5]
2. HyperQuicksort [5]
3. Sanders QuickSort [9]

4.3 Sample Sort

Sample Sort is a splitter-based parallel sorting algorithm. This algorithm determines a load balanced splitting of keys by sampling the global key set. All $p-1$ splitters are typically computed simultaneously, so only one round of all-to-all data movement is required to place the elements on the proper destination processors.

4.3.1 Terminology

Some additional terms ought to be defined specifically for Sample Sort.

- s - length of the sample drawn on each processor.
- $Samp_{[1,p]}$ - sample sequences drawn on each processor.
- $FSamp$ - a combined sample containing the merged sequences $Samp_{[1,p]}$, with length $s \times p$.
- $Sptr_{[1,p-1]}$ - splitter keys designed to partition the entire key set into p approximately equal-sized, continuous chunks.
- $Send_{[1,p],[1,p]}$ - sequences to send; $Send_{i,j}$ is the sequence of elements to be sent from processor i to processor j. $Send_{i,j}$ is contained in both Π_i and Ξ_j.

4.3.2 Sample Sort structure

Below we provide a common Sample Sort algorithm structure similar to [10]. The given procedure with sample size $s = p - 1$.

1. Each processor k sorts its local sequence, Π_k.
2. Each processor k extracts a local sample $Samp_k$ of length s. The local sample is defined such that for $i \in [1, s]$ $Samp_k[i] = \Pi_k[len_k \times \frac{i}{s}]$.
3. The local samples $Samp_k$ for $k \in [1, p]$ are collected and merged into sample $FSamp$ of length $s * p$.
4. Splitter vector $Sptr$ of length $p-1$ is extracted from the combined sample, $FSamp$. The splitters are selected such that $Sptr[j] = FSamp[(p-1) \times (\frac{j}{s} - \frac{1}{2})]$.
5. Each processor k extracts sequences $Send_{k,j}$ for $j \in [1, p]$ from Π_k. $Send_{k,j}$ for $j \in [2, p-1]$ contains all elements $elem$ in Π_k, where $Sptr[j-1] \leq elem < Sptr[j]$. $Send_{k,1}$ and $Send_{k,p}$ contain elements such that $elem < Sptr[0]$ and $elem > Sptr[p-1]$, respectively.
6. The $Send_{k,j}$ sequences are transposed so that each processor k gets sequences $Send_{j,k}$ for $j \in [1, p]$.
7. Each processor k merges the now local sequences $Send_{j,k}$ for $j \in [1, p]$, forming the final sorted sequences Ξ_k.

4.3.3 Resolution of forces

1. Load Balance

   Resolution: Good (4/5)

   Explanation: The Regular Sample Sort detailed above has a theoretical load imbalance upper bound of $\frac{2n}{p}$ if each processor has $\frac{n}{p}$ elements at the start [11]. In practice, the algorithm will usually achieve almost perfect load balance. Other variants of Sample Sort (e.g.

Random Sample Sort [cite]), are not as well load balanced and typically require over-sampling ($s > p$) to achieve a low enough *threshold*.

2. Data movement

   Resolution: Excellent (5/5)

   Order of growth: $\Theta(n)$

   Explanation: Data movement in Sample Sort is minimal, since each element is guaranteed to migrate at most once. By defining all the splitters at once, a single round of communication is sufficient for data movement.

3. Communication latency

   Resolution: Moderate (3/5)

   Order of growth: $\Theta(p^2)$

   Explanation: This algorithm needs at worst p^2 messages to move all the data, and this is not optimal. However, aside from data movement, Sample Sort uses little supplementary latency.

4. Supplementary communication bandwidth

   Resolution: Good (4/5)

   Order of growth: $\Theta(s \times p)$

   Explanation: The sample collection can become a problem on very large systems since it scales with the order of $\Theta(p^2)$.

5. Ability to exploit partially sorted initial distribution

   Resolution: Moderate (3/5)

   Explanation: The data migration generally happens only if it is necessary. However, the sample extraction and collection is not adaptive to distribution.

6. Overlapping communication with computation

   Resolution: Good (4/5)

   Explanation: Sample Sort can exploit overlap of communication with computation by merging incoming data while the data movement is happening. On a large system, some chunks of data wil arrive at their destination processor much sooner than others. Therefore, merging chunks that arrive earlier offsets a lot of the communication overhead.

   Unfortunately, in the Regular Sampling version of Sample Sort, overlap between the sampling communication and computation is difficult or impossible to achieve. This difficulty exists because the local data needs to be sorted completely before the sample is selected. Other versions, of Sample Sort, namely Sample Sorting by Random Sampling do not impose this criterion and can achieve more overlap.

4.3.4 Other Sample Sort algorithms variants

1. Random Sampling [12]
2. Helman-Bader-JaJa Sort [13]

4.4 Histogram Sort

Histogram Sort [8] is a splitter-based parallel sorting algorithm. This sorting algorithm determines a load balanced splitting of keys by *iteratively adjusting a probe of splitter-guesses.* Once all $p-1$ splitters are found within a threshold number of keys $t_{sp} = \frac{1}{2}threshold$, only one round of all-to-all data movement is required to place the elements on the proper destination processors.

4.4.1 Terminology

Some additional terms ought to be defined specifically for Histogram Sort.

- r_b - key minimum.
- r_e - key maximum.
- Prb - a sequence of key values that serve as splitter-guesses.
- pb - length of probe sequence (number of splitter-guesses).
- $Hist_{[1,p]}$ - local histograms containing the number of keys fitting into key ranges generated by a probe Prb. Each $Hist_k$ is of length $pb + 1$.
- $FHist$ - a global histogram of size $pb+1$; a sum of the p $Hist$ sequences.
- $Sptr_{[1,p-1]}$ - splitter keys designed to partition the entire key set into p approximately equal-sized, continuous chunks.
- $Send_{[1,p],[1,p]}$ - sequences to send; $Send_{i,j}$ is the sequence of elements to be sent from processor i to processor j. $Send_{i,j}$ is contained in both Π_i and Ξ_j.

4.4.2 Histogram Sort structure

The basic Histogram Sort algorithm structure, first proposed in [cite Krish93], is below. The given procedure operates on p processors with probe size $pb = p - 1$. Additionally, it is assumed that all keys are in some finite range $[r_b, r_e]$; this assumption is not required by Histogram Sort but simplifies the algorithm.

1. Each processor k sorts its local sequence, Π_k.
2. Probe Prb of size pb is defined uniformly over the range so that $Prb[i] = r_b + (r_e - r_b) \times \frac{i}{pb+1}$.
3. Each processor k computes a histogram $Hist_k$ of size p by calculating how many local keys are smaller than each splitter-guess in the probe. More precisely, for $j \in [2, p-1]$, $Hist_k[j]$ is the number of keys in Π_k which satisfy, $key < Prb[j]$. $Hist_k[1]$ and $Hist_k[p]$ are, respectively, the counts of keys which satisfy $key < Prb[1]$ and $key < r_e$.
4. The local histograms $Hist_k$ for $k \in [1, p]$ are collected and summed into a global histogram $FHist$ of length p.

5. For each splitter, $FHist$ can be used to bound future guesses by or set the splitter to a corresponding Prb key. $Prb[i]$ is a satsifactory splitter $Sptr[j]$ for $i, j \in [1, p-1]$ if $|FHist[i] - \frac{n}{p} \times j| < t_{sp}$. The bounds can be taken as $Prb[k]$ and $Prb[k+1]$, such that $FHist[k] < \frac{n}{p} \times j < FHist[k+1]$.

6. Until all splitters are satisfied, new probes Prb are produced using updated splitter bounds and steps 3-6 are repeated.

7. Each processor k extracts sequences $Send_{k,j}$ for $j \in [1, p]$ from Π_k. $Send_{k,j}$ for $j \in [2, p-1]$ contains all elements $elem$ in Π_k, where $Sptr[j-1] \leq elem < Sptr[j]$. $Send_{k,1}$ and $Send_{k,p}$ contain elements such that $elem < Sptr[1]$ and $elem > Sptr[p-1]$, respectively.

8. The $Send_{k,j}$ sequences are transposed so that each processor k gets sequences $Send_{j,k}$ for $j \in [1, p]$.

9. Each processor k merges the now local sequences $Send_{j,k}$ for $j \in [1, p]$, forming the final sorted sequences Ξ_k.

4.4.3 Resolution of Forces

1. Load balance

   Resolution: Excellent (5/5)

   Explanation:

   The iterative guessing technique used by Histogram Sort is generally more adaptive and cheap than the one-time sampling phase done by the similar Parallel Sample Sort. An arbitrarily defined *threshold* (load imbalance limit) can be achieved using the algorithm.

   The histogramming technique also acheives improved parallel scalability since it requires only a histogram of size $\Theta(p)$, as compared to a sample of size $\Theta(p^2)$ seen in Parallel Sample Sort. The number of histogramming iterations is small if a suitable algorithm is used for probe generation.

   The probe generation algorithm is probably best implemented by keeping track of a possible range for every splitter and placing splitter-guesses as in a binary search. Such an algorithm is reliable since binary search has a good worst-case performance bound. The worst-case performance is important since iterations continue until the very last splitter is determined.

2. Data movement

   Resolution: Excellent (5/5)

   Order of growth: $\Theta(n)$

   Explanation: Data movement in Histogram Sort is minimal, since each element is guaranteed to migrate at most once. All data movement happens in one round.

3. Communication latency

   Resolution: Moderate (3/5)

   Order of growth: Θp^2

   Explanation: This algorithm needs at most p^2 messages to move all the data, and this is not optimal.

   The additional latency required for histogramming (typically a reduction and broadcast every operation) is only a bottleneck if the number of processors is extremely large or many histogramming iterations are required. Neither of these cases is common.

4. Supplementary communication bandwidth

   Resolution: Good (4/5)

   Order of growth: Θp

   Explanation: In the rare case that $n < p$, the histogramming communication becomes the bottleneck. Each histogramming step only requires a single collection operation (reduction) and a broadcast both of size pb which is typically close to p.

5. Ability to exploit partially sorted initial Distributions

   Resolution: Good (4/5)

   Explanation: If the initial splitter guesses are good (uniform distribution or use of splitters from previous sorting iterations), the number of histogramming iterations is small. Also, the data migration of keys generally happens only if it is necessary. This fact is evident since data only migrates from its initial processor to its destination processor.

6. Overlapping communication with computation

   Resolution: Excellent (5/5)

   Explanation: Like Sample Sort, Histogram Sort can exploit overlap of communication with computation by merging incoming data while the data movement is happening. Merging chunks that arrive earlier than others offsets a lot of the communication overhead.

   Histogram Sort also has the potential to overlap the initial sorting work with the histogramming communication. In fact, for certain problem sizes, most of the histogramming work and communication can be effectively removed from the critical path. To achieve this overlap, it is necessary to use a 'splicing technique' for accumulating local histograms. This technique as well as other optimizations to Histogram Sort are detailed in [14].

4.4.4 Histogram Sort Variants

Histogram Sort can be modified to act as a purely comparison based sort [8]. Several extensions for improving the performance and scalability of Histogram Sort are discussed in [14].

4.5 Radix Sort

4.5.1 Overview of Parallel Radix Sorting

Radix Sort [12] is a counting-based sorting algorithm using bitwise representation of keys to obtain a proper ordering. In radix sort algorithms, numbers are iteratively re-arranged into *buckets* by successively considering each digit of the keys

based on the bit representation of keys. The name "radix sort" comes from the fact that the number of buckets that the keys are grouped into is based on the *radix*, or "digit" representation, of the numbers. For example, the radix of a decimal number is 10, the radix of a binary number is 2, and the radix of octal numbers is 8. Parallel radix sort depends on having bounded length keys, and works best when the length of the keys is small(e.g. 32-bit or 64-bit).

Radix sort is parallelized by associating each radix bucket with a processor. The set of keys are partitioned into these buckets. Parallelization of radix sort can be done by assigning each processor a bucket corresponding to a digit value of the current digit examined. For example, if we were sorting numbers using radix 10, and we had a 5 processor machine available, then the first processor would get keys with digit value 0 and 1, the second processor would get keys with digit value 2 and 3, and so on.

In each iteration, the parallel prefix sum computation [15] determines the correct position of the element within the bucket. Upon determining the bucket, each processor sends or copies that key to the appropriate bucket. Once all processors have copied elements from their bucket to new buckets (as necessary), the next iteration proceeds using the next r bits of the key. The proof that this parallel radix sort is correct is beyond the scope of this paper, but one can consult [6] for more details.

4.5.2 Definitions

In parallel radix sort, we use the following terminology:

- b - the number of bits used to represent each of the n keys.
- r - the number of bits used for the radix. Note that $r <= b$.
- i - identifies a digit of a key. Assuming radix r, the digit i is composed of bits in the range $[i \times r, (i+1) \times r)$ of that key's bit representation.
- $Bucket_{[1,2^r]}$ - the array of buckets.
- $Bucket\_send(k)_{[1,2^r]}$ - the array of buckets containing keys sent by processor k.
- $Buckets\_assignments_{[1,2^r]}$ - an array of integers indicating the processor number that each bucket is currently assigned to.
- $Key\_counts(k)_{[1,2^r]}$ - the counts of keys (histogram of key distributions) of the kth processor.
- $Key\_counts_{[1,2^r]}$ - the counts of keys(histogram of key distributions) over all processors.

4.5.3 Algorithm

Below we present a basic example of parallelized radix sort. Note that each of the 2^r buckets is assigned to one of the $|P|$ processors. The assignments are initialized in $Buckets\_assignments\_[1, 2^r]$.

for(i = 0; i < b/r - 1 ; i++)

1. Each processor puts its keys in one of the 2^r buckets, based on the value of the bits (of its keys) in the range $[i \times r, (i+1) \times r)$. With this, the key counts (local histograms) within each processor k are generated and stored in $Key\_counts(k)_{[1,2^r]}$.
2. The counts of keys within each of the 2^r buckets in $Buckets_{[1,2^r]}$ is obtained over all processors. These key counts within each processor $Key\_counts(k)_{[1,2^r]}$ are accumulated in $Key\_counts_{[1,2^r]}$ and broadcasted to all processors. With this information, processors populate $Bucket\_send(k)_{[1,2^r]}$ in order to send their keys to the appropriate processor.
3. Based on the global counts of keys $Key\_counts_{[1,2^r]}$, buckets are redistributed to processors accordingly. The buckets are assigned to processors in continuous chunks in a way that optimizes for load balance. The assignment of processors to buckets is updated within the array $Buckets\_assignment\_[1, 2^r]$.
4. Based on global counts of keys $Key\_counts_{[1,2^r]}$ and $Buckets\_assignment\_[1, 2^r]$, each processor sends keys to the appropriate destination processors. The keys that processor k sends to other processors can be identified by $Bucket\_send(k)_{[1,2^r]}$.

4.5.4 Resolution of forces

1. **Load Balance**

   Resolution: Good (4/5)

   The presented parallel radix sort algorithm is reasonably load balanced because the assignment of buckets to processors is controlled using the sum reduction in order to equalize the number of keys assigned to each processor.

2. **Data movement**

   Resolution: Moderate (3/5)

   Order of growth: $\Theta(k \times n)$

   Explanation: In each of the $k = \frac{b}{r}$ iterations of radix sort, every processor communicates a "disjoint subset of keys to every other processor", known as all-to-all personalized communication [15].

   If the initial distribution is completely random, then in each iteration of parallel radix sort, $\frac{p-1}{p}$ of all $\frac{n}{p}$ elements on a particular processor are moved to a different processor and $\frac{1}{p}$ of all elements remain on that same processor.

   There are a constant number k data redistribution phases(where $k > 0$), and this results in expected total data movement on the order of $\Theta(k \times n)$.

3. **Communication latency**

   Resolution Moderate (3/5)

   Order of growth: $\Theta(k \times p^2)$

   Explanation: Each of the k iterations involves an all-to-all personalized communication amongst p processors to redistribute keys to their respective processors.

This collective communication step yields an undesirable p^2 communication latency at each of the k iterations. It should be noted that performance degrades with large-size keys, especially when $k \geq 64$. Performance also degrades very dramatically for a large number of processors.

4. **Supplementary communication bandwidth**

   Resolution: Moderate (3/5)

   Order of growth: $\Theta(k \times 2^r)$

   Explanation: The only supplementary communication required by Radix Sort is the summation (typically a sum reduction) of Key_counts, which has $Theta(2^r)$ values and the broadcast of $Buckets\_assignments$ which has $Theta(2^p)$ values. The broadcast is generally much cheaper than the data movement but the reduction can become a bottleneck if the number of data elements is small and the radix is large.

5. **Ability to exploit partially sorted initial distribution**

   Resolution: Poor (1/5)

   Explanation: Even in a perfectly sorted data set, keys will be moved around in intermediate iterations before they return (in the last phase) to the processor they were on originally. Thus, radix sort does not exploit the initial distribution well.

6. **Overlapping Communication with Computation**

   Resolution: Moderate (3/5)

   Explanation: Radix Sort can exploit some overlap between communication and computation by calculating counts during the data movement using incoming data. The algorithm could also potentially exploit overlab by packaging the data into buckets while the counts are accumulated.

   However, the total potential for overlap is rather limited in practice and contains an overhead. For example, it can be faster to put keys into buckets while calculating the local counts.

4.5.5 Discussion

Particularly on a shared memory machine with a standard invalidate-based cache coherence protocol, radix sort's irregular sharing pattern can be a drawback, as this can create significant number of cache misses on each iteration. This indicates an irregular sharing pattern between processors: no two processors communicate in a well-defined pattern across the iterations of the algorithm. On a shared memory machine, Radix Sort's inherent irregular sharing pattern causes it to be cache inefficient. Each key may need to go to any one of the log(r) buckets each iteration, independent of which bucket it resided in the previous iteratio. It is important to observe that over all the iterations of radix sort, each processor has an equal number of elements in its bucket.

Radix sort depends on the bit-wise representation of key to be of bounded length. Examples of data types with bounded length keys are integers, characters and floating point numbers. If the keys are 32-bit or 64-bit long, then the number of phases is small and known, and so radix can be used effectively. If the key is a string of unbounded length(such as a person's name), then radix sort will not be effective. This is because the number of iterations can be very large, and a preprocessing pass is needed to figure out the length of the largest size key.

4.5.6 Other examples of Parallel Radix Sort

1. Thearling's radix sort [16]
2. Sohn's Load balanced Parallel Radix Sort [17]

4.6 Summary of Parallel Sorting Solutions

In order to show a side-by-side comparison of the sorting solutions, we summarize our assessment of forces in Table 4.6. Note that this table can be expanded further to include more algorithms as well as more forces.

5. EXAMPLES

1. **Barnes-Hut Morton ordering**: In n-body simulations, one approach to decompose particles accross processors is to use the Morton ordering [18]. For simplicity, assume that coordinates are normalized between 0 and 1. Morton ordering is obtained by creating a key for each particle, by interleaving its x , y, and z coordinates, and sorting particles based on this key. This ensures that nearby bodies are in Morton order(also called Z-order) are likely to be nearby in geometric space. For example, looking at the first three bits it is clear that all bodies in the first octant (whose key begins with 000) occur before any key in the second octant(whose keys begin with 001). By decomposing particles in this the linearized order, one can reduce communcation costs in the actual Barnes-Hut application.

   Sorting particles are done using Z-order, interleaving bits of x, y and z coordinates. N-body applications typically involve a sequence of timesteps, with the sorting and tree-building has to be done at every timestep. Particles don't move that much over one timestep, so the bodies are already mostly sorted. Thus, the force for exploiting initial distribution is strong for this application. This rules out radix sort. Both sample Sort and Quicksort lead to few keys being moved.

   The load balance of this application is impacted by end distribution obtained by the sorting algorithm. This argues in favor of parallel quicksort, which ensures perfect load balance at the end. In contrast, sample sort may lead to twice as many keys as the average on some processor. Even so, quicksort has a large communication cost, which might push us to select parallel sampleSort, especially when the number of processors or the number of particles is large.

2. **PageRank** In Google's PageRank [19] algorithm, webpages are ranked according to their relevance to the query, and then must be sorted efficiently in parallel. The number of pages is huge, and these pages are gathered from multiple servers of Google. Among the

Algorithm	Load Balance	Data Movement	Comm Latency	Add'l Bandwidth	Exploitation of Init Dist	Overlapping Comm with Comp
Quicksort	Good	Good	Excellent	Good	Good	Moderate
Sample Sort	Good	Excellent	Moderate	Good	Moderate	Good
Histogram Sort	Excellent	Excellent	Moderate	Good	Good	Excellent
Radix Sort	Good	Moderate	Moderate	Good	Poor	Moderate

Table 1: Summary of Forces Assessment for Presented Parallel Sorting Solutions

solutions we presented, quicksort would be best because only the top few documents is needed for showing the first screen to the user. Quicksort formulation allows us to use more processors to sort the higher-ranked portion of the data. In addition, if we choose to ignore(omit/delay) sorting of lower-ranked pages, the communication overhead is reduced. Thus, the higher communication cost of quicksort may not be a serious problem. On the other hand, sample sort and radix sort must run to completion before we can find the top elements.

3. **Sorting ASCII codes** Often times, machines must implement a sorting algorithm directly in hardware [5] for sorting a large number of characters and numbers that use a standard bit representation the codes. Radix sort is often times useful for very fast sorting of numbers in hardware. Radix sort depends on having a bounded size of the keys. Unlike quicksort or sample sort which take $\Theta\frac{n}{p} \times log\frac{n}{p}$ total time for computation, parallel radix sort takes only O(kn/ p) time where k is the number of "digits" used in the radix. While parallel radix may incur somewhat more communication overhead than quicksort(as shown in solutions section), this is alleviated by the improvement from log-linear to linear complexity that we get from using radix sort.

   Of course, if the key is a string of unbounded length (such as a name), then radix sort will not be effective. This is because the number of iterations will be too large, and a preprocessing pass is needed to figure out the length of the largest size key. In this case, it is best to use parallel sample sort or parallel quicksort. However, the standard ASCII codes are a finite known length. Thus, since the number of phases is small and known, a parallel radix sort will be most efficient in this case.

6. INVARIANTS

In this section, we present the invariants (i.e. necessary conditions) that must hold during execution of any parallel sorting algorithm. These invariants pertain primarily to the properties of the set of elements to be sorted. **Preconditions**:

The input is a set of elements $\Pi = \{k_1, k_2, \ldots, k_n\}$ distributed across p processors.

Invariants:

1. All keys in the initial distribution are preserved (we don't lose any keys during the redistribution across processors).
2. The set of keys is partitioned into at least p mutually exclusive subsets(we haven't duplicated any keys).
3. The set of all keys must satisfy the properties of a poset (partially order set).

Post-Conditions:

1. All keys in the initial distribution are preserved (we have not lost any keys)
2. All keys within each processor are sorted in ascending order.
3. $\forall i$ such that $i < p$, the largest key on processor p_i is less than or equal to the smallest key on processor p_{i+1}.
4. The resulting output Ξ should be a sequence of keys that satisfies the properties of *chain* (totally ordered set).

7. KNOWN USES

Radix sorting variants have been around as early as 1920s. In 1925, the IBM 80 Electric Punched Card Sorting Machine was developed for sorting punch cards, and used a method resembling radix sort. Yet, as of this writing, the usage of parallel sorting has yet to mature. We discuss some of the current known uses, but the usage of parallel sorting continues to increase by the day.

Parallel sort serves to be a very crucial and representative benchmark for evaluation of today's multi-core and high-performance machines. For multi-core benchmarking, a parallel radix sort implemented within the SPLASH multi-core benchmark suite is often used [**?**]. For benchmarking of high-performance clusters, the Integer Sort (a variant of parallel bucket sort) from the NAS Parallel benchmark suite is often used.

In computation kernels for genetics at LLNL, sorting ATGC strings (DNA sequences) for gene simulation uses sampleSort with a parallel mergeSort on sublists across nodes, and may use a simpler sort for the base case of sorting within nodes.

At Yahoo!, the process of sorting the relevance of webpages for web searches use a parallel counting sort algorithm. The importance of sorting is important in this context. The TeraSort Benchmark [20] implemented using Hadoop (Yahoo's implementation of MapReduce) has gained wide acceptance as a key benchmark for web information retrieval.

Many Oracle Database systems make heavy use of sorting algorithms, and must be optimized for the type of data it must process. A variant of parallel merge sort has been shown to be optimized for space and disk usage in many of these implementations.

8. RELATED PATTERNS

1. **Master-Worker:** Master worker parallelism pattern involves a master processor managing communication of all processors. The Master-Worker pattern is observed in the parallel sample sort algorithm where a Master processor (usually processor P0) broadcasts its chose pivot values to all other processors. In sample sort, all other processors must send in their pivots to an arbitrarily chosen "master" processor. Then, that master processor to sort the samples. All other processors can continue to do local work after receiving a broadcast of the sample distribution pivots sent by the master processor.

2. **Data Parallelism:** In all sorting algorithms there is a local phase for sorting. The local phase of parallel sorting involves applying the same sorting function across all processors. All the local sorts are independent of each other.

3. **Recursive Splitting:** The recursive splitting pattern involves partitioning data recursively, in such a way that the recursive partition will yield maximum concurrency. The parallel Quicksort algorithms partitions keys by choosing a pivot which will group the data into two parts, and then recursively splitting and groups these two parts. One group of processors manage all numbers keys than the pivot, and the other group of processors manage all keys greater than the pivot. Through a careful choice of pivots, one can maintain load balancing in such a way as to provides maximum concurrency.

Acknowledgments

This work would not be possible without the feedback from Computer Science faculty and students from University of Illinois at Urbana and University of California-Berkeley. We would like to especially thank Ralph Johnson, and Kurt Kuetzer for their input and encouragement.

9. REFERENCES

[1] Adler, M., Byers, J., Karp, R.: Parallel sorting with limited bandwidth. (1995) 129–136

[2] Francis, R.S., Mathieson, I.: A benchmark parallel sort for shared memory multiprocessors. IEEE Trans. Comput. (12) (1988) 1619–1626

[3] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms, second edition (2001)

[4] Grama, A.Y., Kumar, V., Sameh, A.: Scalable parallel formulations of the barnes-hut method for n-body simulations. In: Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing, New York, NY, USA, ACM (1994)

[5] Quinn, M.J.: Parallel computing (2nd ed.): Theory and Practice. McGraw-Hill, Inc., New York, NY, USA (1994)

[6] Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to Parallel Computing: Design and Analysis of Algorithms. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA (1994)

[7] Huang, J., Chow, Y.: Parallel sorting and data partitioning by sampling. In: Proc. Seventh International Computer Software and Applications Conference. (November 1983)

[8] Kale, L.V., Krishnan, S.: A comparison based parallel sorting algorithm. In: Proceedings of the 22nd International Conference on Parallel Processing, St. Charles, IL (August 1993) 196–200

[9] Sanders, P., Hansch, T.: Efficient massively parallel quicksort. In: IRREGULAR '97: Proceedings of the 4th International Symposium on Solving Irregularly Structured Problems in Parallel, London, UK, Springer-Verlag (1997) 13–24

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

[11] Li, X., Lu, P., Schaeffer, J., Shillington, J., Wong, P.S., Shi, H.: On the versatility of parallel sorting by regular sampling. Parallel Comput. **19**(10) (1993) 1079–1103

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

[13] Helman, D.R., Bader, D.A., JáJá, J.: A randomized parallel sorting algorithm with an experimental study. J. Parallel Distrib. Comput. **52**(1) (1998) 1–23

[14] Solomonik, E., Kale, L.V.: Highly Scalable Parallel Sorting. In: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS). (April 2010)

[15] Dusseau, A.C., Culler, D.E., Schauser, K.E., Martin, R.P.: Fast parallel sorting under logp: Experience with the cm-5. IEEE Trans. Parallel Distrib. Syst. **7**(8) (1996) 791–805

[16] Thearling, K., Smith, S.: An improved supercomputer sorting benchmark. In: Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing, Los Alamitos, CA, USA, IEEE Computer Society Press (1992) 14–19

[17] Sohn, A., Kodama, Y.: Load balanced parallel radix sort. In: ICS '98: Proceedings of the 12th international conference on Supercomputing, New York, NY, USA, ACM (1998)

[18] Connor, M., Kumar, P.: Fast construction of k-nearest neighbor graphs for point clouds. IEEE Transactions on Visualization and Computer Graphics (2010)

[19] Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web (1999)

[20] Fineberg, S.A., Mehra, P.: The record-breaking terabyte sort on a compaq cluster. In: WINSYM'99: Proceedings of the 3rd conference on USENIX Windows NT Symposium, Berkeley, CA, USA, USENIX Association (1999) 8–8