A Scalable Parallel Sorting Algorithm for Contemporary Architectures

David R. Cheng* Viral B. Shah† John R. Gilbert‡ Alan Edelman§

March 20, 2009

Abstract

Modern scientic codes often combine numerical methods with combinatorial methods. Sorting, a widely studied problem in computer science, is an important primitive for combinatorial scientic computing. Increasingly, scientic codes target parallel computers. Scientic programming environments such as Matlab, Python, R, and Star-P provide sorting as a built-in function.

We present a parallel sorting algorithm that is close to optimal in computation and communication, along with scaling results for shared and distributed memory architectures. One of the core motivations for developing this code was the lack of a quality scalable parallel sorting library. Our sorting library is available as open source and is written to be compatible with the C++ Standard Template Library. The code is available for download at `http://gauss.cs.ucsb.edu/code/index.shtml`

1 Introduction

We describe the design and implementation of an algorithm for parallel sorting on contemporary architectures. Distributed memory architectures are widely in use today. The cost of communication is an order of magnitude larger than the cost of computation on such architectures. Often, it is not enough to tune existing algorithms. Newer architectures demand a fresh look at the problems being solved and new algorithms to yield good performance. We propose a parallel sorting algorithm which moves a minimal amount of data over the network. Our algorithm is close to optimal in both the computation and communication required. It moves lesser data than widely used sample sorting algorithms, and is computationally a lot more efficient on distributed and shared memory architectures.

Blelloch et al. [1] compare several parallel sorting algorithms on the CM–2, and report that a sampling based sort and radix sort are good algorithms to use in practice. We first tried a sampling based sort, but quickly ran into performance problems. The cost of sampling is often quite high, and sample sort requires a redistribution phase at the end, so that the output has the desired distribution. The sampling process itself requires "well chosen" parameters to yield "good" samples. We noticed that we can do away with both these steps if we can determine exact splitters

*Goldman Sachs. Part of this work was done as part of David Cheng's MS thesis at MIT

†Interactive Supercomputing. Part of this work was done as part of Viral Shah's PhD thesis at UC Santa Barbara

‡UC Santa Barbara

§MIT, Interactive Supercomputing

Algorithm.
Input: A vector v of n total elements, evenly distributed among p processors.
Output: An evenly distributed vector w with the same distribution as v, containing the sorted elements of v.

1. Sort the local elements v_i into a vector v'_i.
2. Determine the exact splitting of the local data:
   (a) Compute the partial sums $r_0 = 0$ and $r_j = \sum_{k=1}^{j} d_k$ for $j = 1 \dots p$.
   (b) Use a parallel select algorithm to find the elements $e_1, \dots, e_{p-1}$ of global rank $r_1, \dots, r_{p-1}$, respectively.
   (c) For each r_j, have processor i compute the local index s_{ij} so that $r_j = \sum_{i=1}^{p} s_{ij}$ and the first s_{ij} elements of v'_i are no larger than e_j.
3. Reroute the sorted elements in v'_i according to the indices s_{ij}: processor i sends elements in the range $s_{ij-1} \dots s_{ij}$ to processor j.
4. Locally merge the p sorted sub-vectors into the output w_i.

Figure 1: Parallel sorting with exact splitters

quickly. Saukas and Song [14] describe a quick parallel selection algorithm. Our algorithm extends this work to efficiently find $p-1$ exact splitters in $O(p \log n)$ rounds of communication.

Our goal was to design a scalable, robust, portable and high performance sorting code which would form a building block for higher level combinatorial algorithms. We built our code using standards based library software such as the C++ STL (Standard Template Library) and MPI [6], which allows us to achieve our goals of scalability, robustness and portability without sacrificing performance. Our code is highly modular, which lets the user replace any stage of the algorithm with platform or application specific routines for higher performance, if need be.

2 Algorithm Description

We have p processors to sort n total elements in a vector v. Assume that the input elements are already load balanced, or evenly distributed over the p processors - this is not a requirement but makes the description and analysis simpler. We rank the processors $1 \dots p$, and define v_i to be the elements held locally by processor i. The *distribution* of v is a vector d where $d_i = |v_i|$. We say v is evenly distributed if it is formed by the concatenation $v = v_1 \dots v_p$, and $d_i \leq \lceil \frac{n}{p} \rceil$ for all i.

We describe our algorithm in Figure 2. We assume the task is to sort the input in increasing order. Naturally, the choice is arbitrary and any other comparison function may be used.

2.1 Local sort

The first step may invoke any local sort applicable to the problem at hand. It is beyond the scope of this study to devise an efficient sequential sorting algorithm, as this problem is very well studied. We simply impose the restriction that the algorithm used here should be identical to the one used for a baseline comparison on a non-parallel computer. Define the computation cost for this algorithm on an input of size n to be $T_s(n)$. Therefore, the amount of computation done by processor i is just $T_s(d_i)$. Since the local sorting must be completed on each processor before the next step can proceed, the global cost is $\max_i T_s(d_i) = T_s(\lceil \frac{n}{p} \rceil)$. For a comparison based sort, this is $O(\frac{n}{p} \lg \frac{n}{p})$.

2.2 Exact splitting

This step is nontrivial, and the main result of this paper follows from the observation that exact splitting over locally sorted data can be done efficiently.

The method used for simultaneous selection was given by Saukas and Song in [14], with two main differences: local ranking is done by binary search rather than partition, and we perform $O(\lg n)$ rounds of communication rather than terminating the selection process earlier. For completeness, the single selection algorithm is described next.

2.2.1 Single selection

First, we consider the simpler problem of selecting just one target, an element of global rank[1] r. The algorithm for this task is motivated by the sequential methods for the same problem, most notably the one given in [2].

Although it may be clearer to define the selection algorithm recursively, the practical implementation and extension into simultaneous selection proceed more naturally from an iterative description. Define an active range to be the contiguous sequence of elements in v_i' that may still have rank r, and let a_i represent its size. Note that the total number of active elements is $\sum_{i=1}^{p} a_i$. Initially, the active range on each processor is the entire vector v_i' and a_i is just the input distribution d_i. In each iteration of the algorithm, a "pivot" is found that partitions the active range in two. Either the pivot is determined to be the target element, or the next iteration continues on one of the partitions.

Each processor i performs the following steps:

1. Let m_i be the median of the active range of v_i'. Broadcast it to all processors.
2. Weigh median m_i by $\frac{a_i}{\sum_{k=1}^{p} a_k}$. Find the weighted median of medians m_m. By definition, the weights of the $\{m_i | m_i < m_m\}$ sum to at most $\frac{1}{2}$, as do the weights of the $\{m_i | m_i > m_m\}$.
3. Find m_m with binary search over the active range of v_i' to determine the first and last positions f_i and l_i it can be inserted into the sorted vector v_i'. Broadcast these two values.
4. Compute $f = \sum_{i=1}^{p} f_i$ and $l = \sum_{i=1}^{p} l_i$. The element m_m has ranks $[f, l]$ in v.

[1] To handle the case of non-unique input elements, any element may actually have a range of global ranks. To be more precise, we want to find the element whose set of ranks contains r.

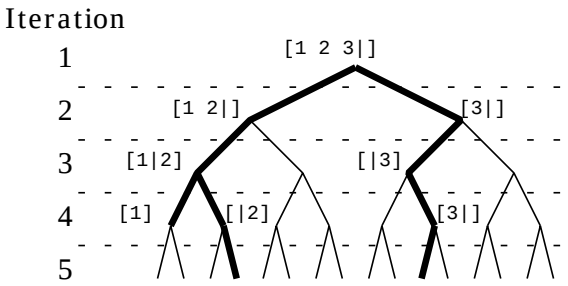

Figure 2: Example execution of selecting three elements. Each node corresponds to a contiguous range of v'_i, and gets split into its two children by the pivot. The root is the entire v'_i, and the bold traces which ranges are active at each iteration. The array at a node represents the target ranks that may be found by the search path, and the vertical bar in the array indicates the relative position of the pivot's rank.

5. If $r \in [f, l]$, then m_m is the target element and we exit. Otherwise the active range is truncated as follows:

   Increase the bottom index to $l_i + 1$ if $l < r$; or decrease the top index to $f_i - 1$ if $r < f$.

   Loop on the truncated active range.

We can think of the weighted median of medians as a pivot, because it is used to split the input for the next iteration. It is a well-known result that the weighted median of medians can be computed in linear time [5, 13]. One possible way is to partition the values with the (unweighted) median, accumulate the weights on each side of the median, and recurse on the side that has too much weight. Therefore, the amount of computation in each round is $O(p) + O(\lg a_i) + O(1) = O(p + \lg \frac{n}{p})$ per processor.

Furthermore, as shown in [14], splitting the data by the weighted median of medians will eliminate at least $\frac{1}{4}$ of the elements. Because the step begins with n elements under consideration, there are $O(\lg n)$ iterations. The total single-processor computation for selection is then $O(p \lg n + \lg \frac{n}{p} \lg n) = O(p \lg n + \lg^2 n)$.

The amount of communication is straightforward to compute: two broadcasts per iteration, for $O(p \lg n)$ total bytes being transferred over $O(\lg n)$ rounds.

2.2.2 Simultaneous selection

The problem is now to select multiple targets, each with a different global rank. In the context of the sorting problem, we want the $p-1$ elements of global rank $d_1, d_1 + d_2, \ldots, \sum_{i=1}^{p-1} d_i$. One simple way to do this would call the single selection problem for each desired rank. Unfortunately, doing so would increase the number of communication rounds by a factor of $O(p)$. We can avoid this inflation by solving multiple selection problems independently, but combining their communication. Stated another way, instead of finding $p-1$ paths one after another from root to leaf of the binary search tree, we take a breadth-first search with breadth at most $p-1$ (see Figure 2).

To implement simultaneous selection, we augment the single selection algorithm with a set A of active ranges. Each of these active ranges will produce at least one target. An iteration of the algorithm proceeds as in single selection, but finds multiple pivots: a weighted median of medians

for each active range. If an active range produces a pivot that is one of the target elements, we eliminate that active range from A (as in the leftmost branch of Figure 2). Otherwise, we examine each of the two partitions induced by the pivot, and add it to A if it may yield a target. Note that as in iterations 1 and 3 in Figure 2, it is possible for both partitions to be added.

In slightly more detail, we handle the augmentation by looping over A in each step. The local medians are bundled together for a single broadcast at the end of Step 1, as are the local ranks in Step 3. For Step 5, we use the fact that each active range in A has a corresponding set of the target ranks: those targets that lie between the bottom and top indices of the active range. If we keep the subset of target ranks sorted, a binary search over it with the pivot rank[2] will split the target set as well. The left target subset is associated with the left partition of the active range, and the right sides follow similarly. The left or right partition of the active range gets added to A for the next iteration only if the corresponding target subset is non-empty.

The computation time necessary for simultaneous selection follows by inflating each step of the single selection by a factor of p (because $|A| \leq p$). The exception is the last step, where we also need to binary search over $O(p)$ targets. This amount to $O(p + p^2 + p \lg \frac{n}{p} + p + p \lg p) = O(p^2 + p \lg \frac{n}{p})$ per iteration. Again, there are $O(\lg n)$ iterations for total computation time of $O(p^2 \lg n + p \lg^2 n)$.

This step runs in $O(p)$ space, the scratch area needed to hold received data and pass state between iterations.

The communication time is similarly inflated: two broadcasts per round, each having one processor send $O(p)$ data to all the others. The aggregate amount of data being sent is $O(p^2 \lg n)$ over $O(\lg n)$ rounds.

2.2.3 Producing indices

Each processor computes a local matrix S of size $p \times (p+1)$. Recall that S splits the local data v'_i into p segments, with $s_{k0} = 0$ and $s_{kp} = d_k$ for $k = 1 \ldots p$. The remaining $p - 1$ columns come as output of the selection. For simplicity of notation, we briefly describe the output procedure in the context of single selection; it extends naturally for simultaneous selection. When we find that a particular m_m has global ranks $[f, l) \ni r_k$, we also have the local ranks f_i and l_i. There are $r_k - f$ excess elements with value m_m that should be routed to processor k. In order to get a stable sorting algorithm, we assign s_{ki} from $i = 1$ to p, taking as many elements as possible without overstepping the excess. More precisely,

$$s_{ki} = \min \left\{ f_i + (r_k - f) - \sum_{j=1}^{i-1} (s_{kj} - f_j),\ l_i \right\}$$

The computation requirements for this step are $O(p^2)$ to populate the matrix S; the space used is also $O(p^2)$.

2.3 Element routing

The minimum amount of communication in a parallel sorting algorithm involves moving elements from the locations they start out to where they eventually belong (in the sorted order). An optimal

[2] Actually, we binary search for the first position f may be inserted, and for the last position l may be inserted. If the two positions are not the same, we have found at least one target.

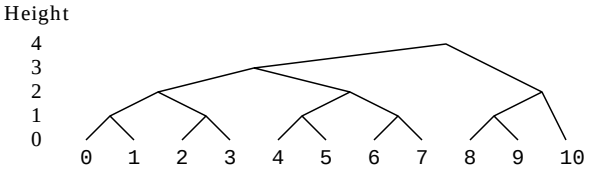

Figure 3: An example of tree merging when the number of processors is not a power of 2.

parallel sorting algorithm will communicate every element from its current location to a location in remote memory at most once. Our algorithm is optimal in this sense. For instance, if the input is already sorted, no data movement occurs. However, if the input is in reverse sorted order, almost all elements may need to be communicated to their destinations. This amount of data communicated in this element routing step is $\theta(n)$.

2.4 Merging

Now each processor has p sorted sub-vectors, and we want to merge them into a single sorted sequence. The simple approach we take for this problem is to conceptually build a binary tree on top of the vectors. To handle the case of p that are not powers of 2, we say a node of height i has at most 2^i leaf descendants, whose ranks are in $[k \cdot 2^i, (k+1) \cdot 2^i)$ for some k (Figure 3). It is clear that the tree has height $\leq \lceil \lg p \rceil$.

For reasons of cache efficiency, we merge pairs of sub-vectors out-of-place from this tree. Cache oblivious algorithms may yield better performance across a variety of architectures. We refer the reader to the literature on cache-oblivious data structures and algorithms [3, 8].

Notice that a merge will move a particular element exactly once (from one buffer to its sorted position in the other buffer). Furthermore, there is at most one comparison for each element move. Finally, every time an element gets moved, it goes into a sorted sub-vector at a higher level in the tree. Therefore each element moves at most $\lceil \lg p \rceil$ times, for a total computation time of $d_i \lceil \lg p \rceil$. Again, we take the time of the slowest processor, for computation time of $\lceil \frac{n}{p} \rceil \lceil \lg p \rceil$.

2.5 Theoretical performance

Let the total computation time $T_s^*(n,p) = \frac{1}{p} T_s(n)$ for $1 \leq p \leq P$. This is the linear speedup in p over any sequential sorting algorithm with running time $T_s(n)$. We examine the time complexities of each step of the algorithm, which results in the total computation time:

$$T_s^*(n,p) + O(p^2 \lg n + p \lg^2 n) + (\lceil \tfrac{n}{p} \rceil \text{ if } p \text{ not a power of 2}) \tag{1}$$

The total space usage aside from the input is $O(p^2 + \frac{n}{p})$. We will provide proofs of the bounds on theoretical computation and communication in the full version of our paper.

We want to compare this algorithm against an arbitrary parallel sorting algorithm with the following properties:

1. Total computation time $T_s^*(n,p) = \frac{1}{p} T_s(n)$ for $1 \leq p \leq P$, linear speedup in p over any sequential sorting algorithm with running time $T_s(n)$.

2. Minimal amount of cross-processor communication $T_c^*(v)$, the number of elements that begin and end on different processors.

We will not go on to claim that such an algorithm is truly an optimal parallel algorithm, because we do not require $T_s(n)$ to be optimal. However, optimality of $T_s(n)$ does imply optimality of $T_s^*(n,p)$ for $p \leq P$. Briefly, if there were a faster $T'_s(n,p)$ for some p, then we could simulate it on a single processor for total time $pT'_s(n,p) < pT_s^*(n,p) = T_s(n)$, which is a contradiction.

2.6 Computation

We can examine the total computation time by adding together the time for each step, and comparing against the theoretical $T_s^*(n,p)$:

$$\begin{aligned} & T_s(\lceil \tfrac{n}{p} \rceil) + O(p^2 \lg n + p \lg^2 n) + \lceil \tfrac{n}{p} \rceil \lceil \lg p \rceil \\ \leq \; & \frac{1}{p} T_s(n+p) + O(p^2 \lg n + p \lg^2 n) + \lceil \tfrac{n}{p} \rceil \lceil \lg p \rceil \\ = \; & T_s^*(n+p,p) + O(p^2 \lg n + p \lg^2 n) + \lceil \tfrac{n}{p} \rceil \lceil \lg p \rceil \end{aligned}$$

The inequality follows from the fact that $T_s^*(n) = \Omega(n)$.

It is interesting to note the case where a comparison sort is necessary. Then we use a sequential sort with $T_s(n) \leq c\lceil \frac{n}{p} \rceil \lg \lceil \frac{n}{p} \rceil$ for some $c \geq 1$. We can then combine this cost with the time required for merging (Step 4):

$$\begin{aligned} & c\lceil \tfrac{n}{p} \rceil \lg \lceil \tfrac{n}{p} \rceil + \lceil \tfrac{n}{p} \rceil \lceil \lg p \rceil \\ \leq \; & c\lceil \tfrac{n}{p} \rceil \lg(n+p) + \lceil \tfrac{n}{p} \rceil (\lceil \lg p \rceil - c \lg p) \\ \leq \; & c\lceil \tfrac{n}{p} \rceil \lg n + c\lceil \tfrac{n}{p} \rceil \lg(1 + \tfrac{p}{n}) + \lceil \tfrac{n}{p} \rceil (\lceil \lg p \rceil - c \lg p) \\ \leq \; & \frac{cn \lg n}{p} + \lg n + 2c + (\lceil \tfrac{n}{p} \rceil \text{ if } p \text{ not a power of 2}) \end{aligned}$$

With comparison sorting, the total computation time becomes:

$$T_s^*(n,p) + O(p^2 \lg n + p \lg^2 n) + (\lceil \tfrac{n}{p} \rceil \text{ if } p \text{ not a power of 2}) \tag{2}$$

Furthermore, $T_s^*(n,p)$ is optimal to within the constant factor c.

2.7 Communication

We have already established that the exact splitting algorithm will provide the final locations of the elements. The amount of communication done in the routing phase is then the optimal amount. Therefore, total cost is:

$$T_c^*(v) \text{ in 1 round} + O(p^2 \lg n) \text{ in } \lg n \text{ rounds}$$

2.7.1 Space

The total space usage aside from the input is:

$$O\left(p^2 + \frac{n}{p}\right)$$

2.8 Requirements

Given these bounds, it is clear that this algorithm is only practical for $p^2 \leq \frac{n}{p} \Rightarrow p^3 \leq n$. Returning to the formulation given earlier, we have $p = \lfloor n^{1/3} \rfloor$. This requirement is a common property of other parallel sorting algorithms, particularly sample sort (i.e. [1, 15, 12], as noted in [11]).

2.9 Analysis in the BSP Model

A bulk-synchronous parallel computer, described in [16], models a system with three parameters: p, the number of processors; L, the minimum amount of time between subsequent rounds of communication; and g, a measure of bandwidth in time per message size. Following the naming conventions of [10], define π to be the ratio of computation cost of the BSP algorithm to the computation cost of a sequential algorithm. Similarly, define μ to be the ratio of communication cost of the BSP algorithm to the number of memory movements in a sequential algorithm. We say an algorithm is c-optimal in computation if $\pi = c + o(1)$ as $n \to \infty$, and similarly for μ and communication.

We may naturally compute the ratio π to be Equation 2 over $T_s^*(n,p) = \frac{cn \lg n}{p}$. Thus,

$$\pi = 1 + \frac{p^3}{cn} + \frac{p^2 \lg n}{cn} + \frac{1}{c \lg n} = 1 + o(1) \text{ as } n \to \infty$$

Furthermore, there exist movement-optimal sorting algorithms (i.e. [7]), so we compute μ against $\frac{gn}{p}$. It is straightforward to verify that the BSP cost of exact splitting is $O(\lg n \max\{L, gp^2 \lg n\})$, giving us

$$\mu = 1 + \frac{pL \lg n}{gn} + \frac{p^3 \lg^2 n}{n} = 1 + o(1) \text{ as } n \to \infty$$

Therefore the algorithm is 1-optimal in both computation and communication.

Exact splitting dominates the cost beyond the local sort and the routing steps. The total running time is therefore $O(\frac{n \lg n}{p} + \frac{gn}{p} + \lg n \max\{L, gp^2 \lg n\})$. This bound is an improvement on that given by [11], for small L and $p^2 \lg^2 n$. The tradeoff is that we have decreased one round of communicating much data, to use many rounds of communicating little data. Our experimental results indicate that this choice is reasonable.

3 Experimental results

The communication cost of our sorting algorithm is near optimal if p is small. Furthermore, the sequential computation speedup is near linear if $p \ll n$. Notice that the speedup is given with respect to a sequential algorithm, rather than to itself with small p. The intention is that efficient sequential sorting algorithms and implementations can be developed without any consideration for parallelization, and then be simply dropped in for good parallel performance.

We now turn to empirical results, which suggest that the exact splitting uses little computation and communication time.

3.1 Experimental setup

We implemented our parallel sorting algorithm in C++ using MPI [6] for communication. The motivation is for our code to be used as a library with a simple interface; it is therefore templated, and comparison based.

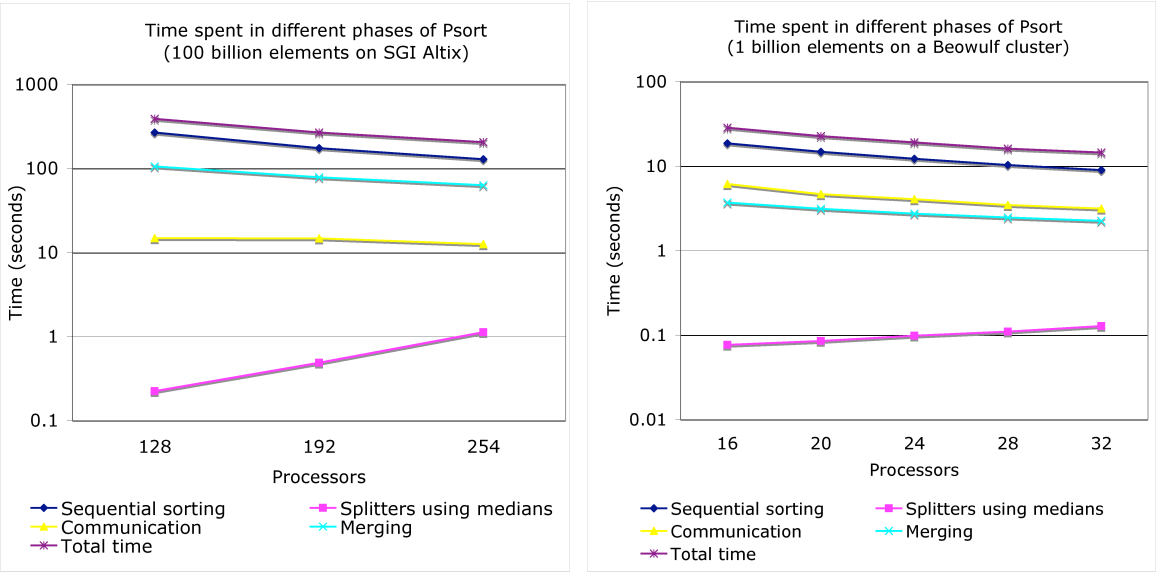

Figure 4: Scalability tests are performed for a fixed problem size, while changing the number of processors. Good scaling is observed on shared memory architectures (left) as well as on clusters (right).

We use `std::sort` and `std::stable_sort` from the C++ Standard Template Library (STL) library for sequential sorting.The C++ STL has one of the fastest general purpose sorting routines available [4].

We use MPI (Message Passing Interface) for communication. It is the most portable and widely used method for communication in parallel computing. Since vendor optimized MPI implementations are available on most platforms nowadays, we expect reasonable performance on distributed as well as shared memory architectures. We use the MPI libraries provided by the SGI MPT on the Altix, and OpenMPI [9] on clusters.

Our choice of the C++ STL sequential sorting routines and MPI allows our code to be robust, scalable and portable without sacrificing performance. We tested our implementation on an SGI Altix and a commodity cluster. The SGI Altix had 256 Itanium 2 processors and 4TB of RAM in a single system image. The beowulf cluster had 32 Xeon processors and 3 GB of memory per node connected via gigabit ethernet.

We run every test instance twice, timing only the second invocation. As a result, setup time such as page table initializations etc. are not counted in our timings.

Figure 3.1 shows the scaling of our algorithm as the same number of elements are sorted on different numbers of processors. We do not provide comparison to sequential performance because the datasets do not fit on any single processor. The largest problem we solved is sorting 100 billion elements with 254 processors in under 4 minutes. Figure 5 shows the scaling of our algorithm with the problem size, the number of processors being fixed. In all the cases, we observe very good scaling on very large problem sizes on a shared memory Altix as well as on a cluster.

For clusters, we also present sequential speedup in Figure 6. For small problems, we do not observe good scaling with small problem sizes. As the problems get larger, we observe better scaling as expected. For the largest problem size (1 billion), it is not possible to run the code on

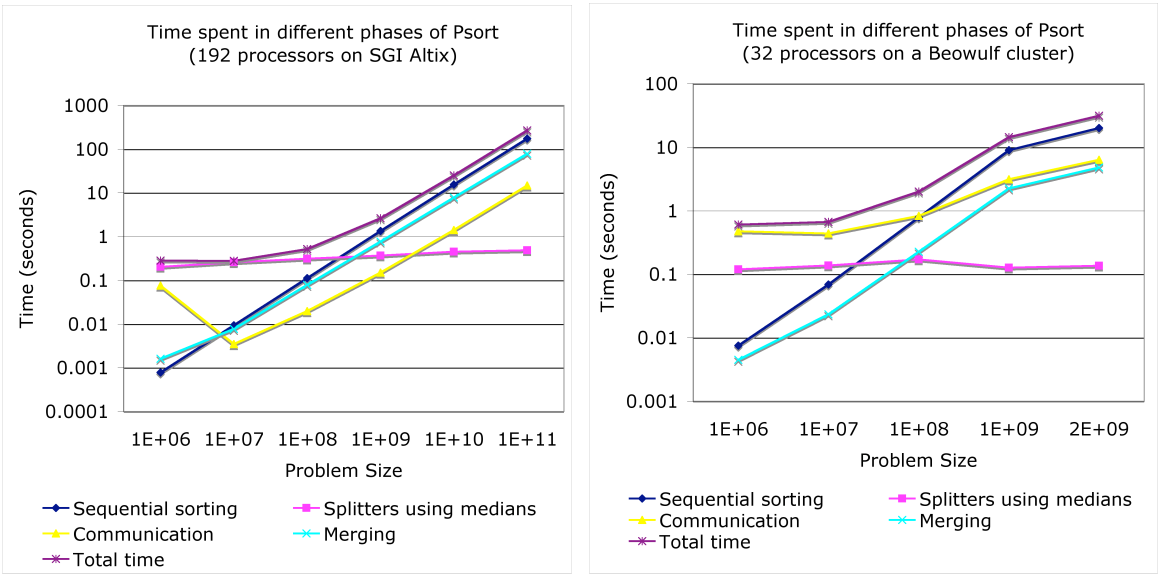

Figure 5: Scalability tests are performed for a fixed problem size while changing the number of processors. Good scaling is observed on large problems on shared memory architectures (left) as well as on clusters (right).

small numbers of processors. We extrapolate the performance for small numbers of processors, and present actual performance for 16 processors and higher.

We also experimented with cache oblivious strategies. We tried using funnel sort [4] for sequential sorting, but found it to be slower than the STL sorting algorithms. On the other hand, a funnel merge did yield slightly better performance than the out-of-place tree merge we use in our code. We compare the performance of the two merging algorithms on the Altix in Figure 7

3.2 Comparison with Sample sorting

Several prior works [1, 12, 15] conclude that sample sort is the most efficient parallel sorting algorithm for large n and p. Such algorithms are characterized by having each processor distribute its $\lceil \frac{n}{p} \rceil$ elements into p buckets, where the bucket boundaries are determined by some form of sampling. Once the buckets are formed, a single round of all-to-all communication follows, with each processor i receiving the contents of the ith bucket from everybody else. Finally, each processor performs some local computation to place all its received elements in sorted order.

One of the problems we encountered with sample sorting was the cost of picking samples, and picking splitters from those samples. Since we are interested in sorting extremely large amounts of data, the sampling step and picking splitters turns out to be very expensive.

The other drawback of sample sort is that the final distribution of elements may be uneven. Much of the work in sample sorting is directed towards reducing the amount of imbalance, providing schemes that have theoretical bounds on the largest amount of data a processor can collect in the routing. The problem with one processor receiving too much data is that the computation time in the subsequent steps is dominated by this one overloaded processor. Furthermore, some applications require an exact output distribution; this is often the case when sorting is just one part of a multi–

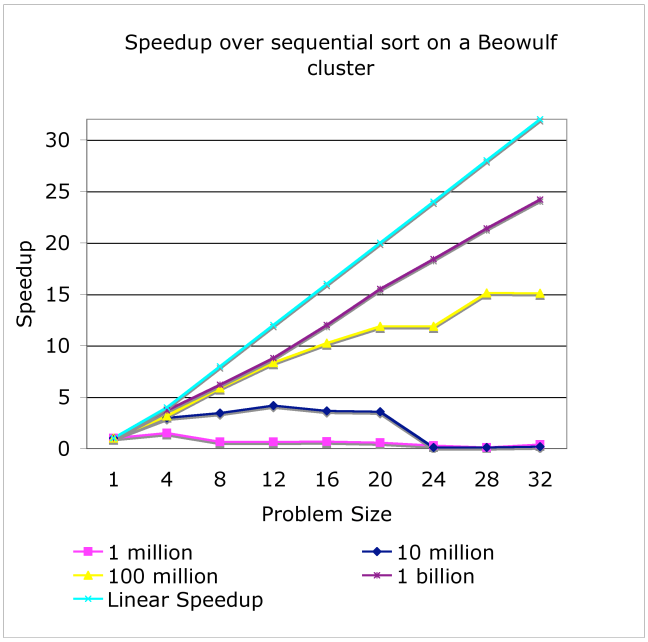

Figure 6: The 45 degree line represents perfect scaling. As the problem size gets larger, better scaling is observed. However, for small to moderate sized problems, the scaling is poor - as expected on beowulf clusters.

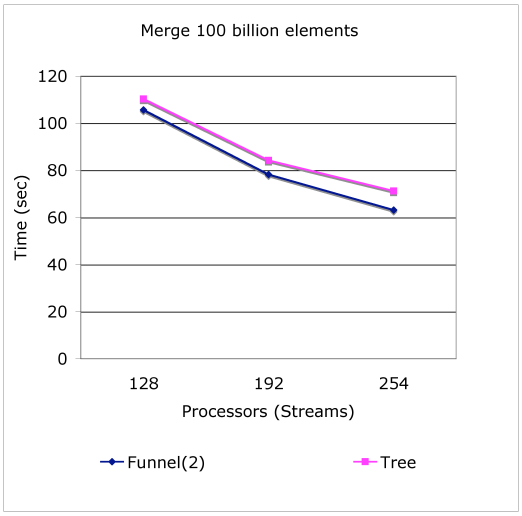

Figure 7: The performance of cache-oblivious merging is compared with simple tree based merging. Cache oblivious outperforms tree merging on very large problem sizes with a large number of processors by up to 15%. The savings are much smaller as a fraction of total sorting time.

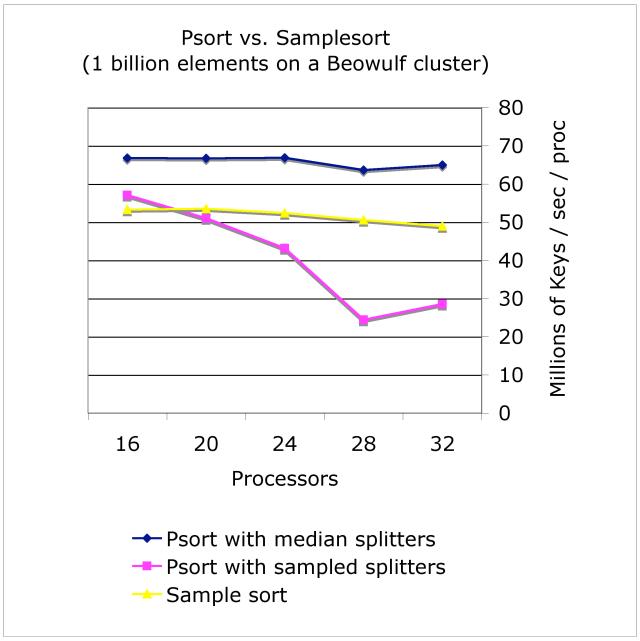

Figure 8: Several parallel sorting algorithms are compared. A parallel sort with exact splitters determined through parallel selection outperforms both implementations of sample sort.

step process. In such cases, an additional redistribution step would be necessary, where elements across the boundaries are communicated.

We compare the performance of our algorithm with two different implementations of sampling based sorts in Figure 8. "Psort with median splitters" is our parallel sorting algorithm which uses medians on each processor to pick exact splitters. "Psort with sampled splitters" is the same algorithm, but it uses random sampling to pick splitters instead of medians. "Sample sort" is the traditional sampling based sorting algorithm, and usually has the following steps:

1. Pick splitters by sampling or oversampling.
2. Partition local data to prepare for the communication phase.
3. Route elements to their destinations.
4. Sort local data.
5. Redistribute to adjust processor boundaries.

The steps in Sample sort differ from the Psort algorithms in two ways. Psort sorts local data first, whereas Sample sort sorts local data as the last step in the algorithm. Sample sort may need to do an extra round of communication to adjust processor boundaries if the resulting distribution is different from the required one.

Our algorithm works much faster than the traditional sample sort although both show good scalability. We do not experiment with a wide range of sampling methods - picking p^2 splitters for p processors. Since our input is uniformly distributed, we do get good splitters. The main performance differences are explained by:

1. Partitioning local data in Sample sort before the element routing step is much slower than merging streams of data received from other processors in Psort. This is because merging is much more cache-friendly than partitioning.

2. Sample sort also has to perform an extra round of communication to re-balance the data distribution, which adds extra penalty to performance. If the sampled splitters do not approximate the distribution well, the load imbalance may be large and this extra round of communication may incur a larger penalty.

4 Conclusion

We present a high performance, highly scalable parallel sorting algorithm which compares favorably against the traditional sample sort algorithm. Our code is uses only the C++ Standard Template Library and MPI, making it robust and portable.

There may be room for further improvement in our implementation. The cost of merging can be reduced by interleaving the p-way merge step with the element rerouting, merging sub-arrays as they are received. Alternatively, using a data structure such as a funnel (i.e. [3, 8]) may allow better cache efficiency to reduce the merging time. Another potential area of improvement is the exact splitting. Instead of traversing search tree to completion, a threshold can be set; when the active range becomes small enough, a single processor gathers all the remaining active elements and completes the computation sequentially. This method, used by Saukas and Song in [14], helps reduce the number of communication rounds in the tail end of the step. Finally, this parallel sorting algorithm will directly benefit from future improvements to sequential sorting and all–to–all communication schemes.

To the best of our knowledge, we have presented a new deterministic algorithm for parallel sorting that makes a strong case for exact splitting on modern high performance computers. Leaving aside some intricacies of determining the exact splitters, the algorithm is conceptually simple to understand, analyze, and implement. Our implementation powers the Star-P `sort` and we hope our efforts will guide other implementations.

References

[1] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 3–16. ACM Press, 1991.

[2] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–460, August 1973.

[3] G. S. Brodal and R. Fagerberg. Funnel heap – A cache oblivious priority queue. In *Proceedings of the 13th International Symposium on Algorithms and Computation*, pages 219–228. Springer-Verlag, 2002.

[4] G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. In L. Arge, G. F. Italiano, and R. Sedgewick, editors, *Proceedings of the Sixth Workshop on*

Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics, pages 4–17. SIAM, 2004.

[5] T. T. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press, 1990.

[6] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, 39(7):84–90, 1996.

[7] G. Franceschini and V. Geffert. An in-place sorting with O(n log n) comparisons and O(n) moves. *Journal of the ACM*, 52(4):515–537, 2005.

[8] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[9] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In D. Kranzlmüller, P. Kacsuk, and J. Dongarra, editors, *PVM/MPI*, volume 3241 of *Lecture Notes in Computer Science*, pages 97–104. Springer, 2004.

[10] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *SPAA*, pages 223–232, 1996.

[11] M. T. Goodrich. Communication-efficient parallel sorting. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*, pages 247–256, 1996.

[12] D. R. Helman, J. JáJá, and D. A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. *J. Exp. Algorithmics*, 3, 1998.

[13] A. Reiser. A linear selection algorithm for sets of elements with weights. *Information Processing Letters*, 7(3):159–162, 1978.

[14] E. L. G. Saukas and S. W. Song. A note on parallel selection on coarse grained multicomputers. *Algorithmica*, 24(3/4):371–380, 1999.

[15] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[16] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.