Communication Conscious Radix Sort[*]

Daniel Jiménez-González Josep-L. Larriba-Pey
Juan J. Navarro

Computer Architecture Dept.
Universitat Politècnica de Catalunya
Jordi Girona 1-3, Campus Nord-UPC, Modul D6, E-08034 Barcelona
e-mail: {djimenez,larri,juanjo}@ac.upc.es

Abstract

The explotation of data locality in parallel computers is paramount to reduce the memory traffic and communication among processing nodes. We focus on the exploitation of locality by Parallel Radix sort.

The original Parallel Radix sort has several communication steps in which one sorting key may have to visit several processing nodes. In response to this, we propose a reorganization of Radix sort that leads to a highly local version of the algorithm at a very low cost. As a key feature, our algorithm performs one only communication step, forcing keys to move only once between processing nodes. Also, it reduces the amount of data communicated. Finally, the new algorithm achieves a good load and communication balance which makes it insensitive to skewed data distributions. We call the new version of Parallel Radix sort that combines locality and load balance, Communication and Cache Conscious Radix sort (C^3-Radix sort).

Our results on 16 processors of the SGI O2000 show that C^3-Radix sort reduces the execution time of the previous fastest version of Parallel Radix sort by 3 times for data sets larger than 8M keys and by almost 2 times for smaller data sets.

1 Introduction

Large scale parallel computers include complex memory hierarchies and communication networks. With the design of software that exploits data locality, both the memory hierarchy and the communication network benefit from lower data traffic.

We address the important problem of locality exploitation for parallel Radix sort. Radix sort is the most competitive parallel in-core sorting algorithm for

[*]This work was supported by the Ministry of Education and Science of Spain under contract TIC-0511/98 and by CEPBA.

large data sets at present. In order to improve the locality of parallel Radix sort we propose a version of this algorithm that we call Communication and Cache Conscious Radix sort, C^3-Radix sort. With C^3-Radix sort, we reduce the amount of communication to one only step. Also, we reduce the amount of data sent by dinamically reassigning Logic processors to real processors during this communication step. However, a moderate increase in the number of operations and load unbalance appear in a first version of our algorithm. While the former is not important because it is compensated by far with the reduction in communication time, the latter turns out to be a problem that we also analyze and control.

C^3-Radix sort makes use of CC-Radix sort which is a sequential version of Radix sort that reduces cache and TLB misses by exploiting locality. This sequential version of Radix sort is analyzed in detail in the companion paper [JiNL98].

To show how important the exploitation of locality is, C^3-Radix sort, reduces by 3 times the execution time of Load Balanced Radix [SoKo98] for data sets larger than 8M keys of 32 bits each and 2 times for smaller data sets on 16 processors of the SGI O2000. To our knowledge, Load Balanced Radix is the fastest parallel in-memory version of Radix sort proposed up to now.

In the field of parallel sorting, efforts have been addressed to solve the problem for disk resident data [DNS92, Nyb94, Aga96, Arp97] and for memory resident data [FrM88, Hel96, Hel96, SoKo98]. Usually, research on disk resident data focusses on minimizing the traffic between the disk and the memory. Thus, those works increase the locality of computations by working on data subsets that fit in the memories of the processing nodes. In some of those papers [Nyb94, Aga96, Arp97] and one of the memory resident papers [Hel96], some memory hierarchy locality issues are addressed. However, to our knowledge, the important issue of optimizing communication has not been adressed explicitely but in [Hel96] where a technique called personalized communication is used to perform an optimal scheduling of communications for Radix sort. This last paper, though, does not focus on reducing the number of steps and amount of communication for Radix sort as we do here. In addition, the algorithm proposed in [SoKo98], that we outperform, improves the execution time of the algorithms presented in [Hel96].

The paper is structured as follows. In Section 2 we explain the basic sequential and parallel versions of Radix sort and discuss their advantages and disadvantages. In Section 3 we explain C^3-Radix sort. In Section 4 we describe the experimental setup that we utilize. Then, Section 5 is devoted to analyze the algorithm and understand its behaviour on the O2000. In Section 6 we outline previous works on the topic. In Section 7 we compare our algorithm to previous parallel sorting implementations. Finally, in Section 8 we conclude.

2 Parallel Radix sort

Let us suppose that we want to sort N integer keys of b bits each. Radix sort starts by dividing the b bits into m digits of b_i consecutive bits where $\sum_{i=0}^{m-1} b_i = b$. For each digit starting from the least significant one, the method performs three steps. First, using the values of digit i, the method builds a histogram of the N keys on 2^{b_i} counters. Then, partial sums of the counters are made in such a way that they can be used as indices to store the N keys into 2^{b_i} buckets in an auxiliary destination vector. Finally, the keys are moved to the corresponding buckets in the destination vector with the help of the indices created previously. This sequential algorithm is described in [Knu73].

There are a couple of remarks that can be made about the nature of the sequential version of Radix Sort

- The memory needed is proportional to $2n$ positions for the keys plus $\max_{i=0}^{m-1} 2^{b_i}$ positions to hold the counters of each group of b_i bits.
- After sorting digit k, we say that the N keys are sorted for the $\sum_{i=0}^{k-1} b_i$ least significant bits.

The distributed parallel version of this algorithm is similar to the sequential one adding some communication steps. Let us suppose that the N keys are distributed evenly accross the P available processors. Initially, each processor performs locally the three sequential steps on the least significant digit of b_0 bits. This initial sequence has perfect load balance. After that, 2^{b_0} buckets are formed in each processor as shown in the example of Figure 1.a for $P = 4$ and $b_0 = 2$. The number of buckets 2^{b_0} may differ from the number of processors although this is not what the example of Figure 1 shows. Now, the objective is to perform an all to all communication so that buckets with equal values for the least significant digit are placed in the same processor or group of processors contiguously as shown in Figure 1.b. After that, the sorting of the b_1 bits of digit 1 starts. This will be repeated for each of the digits of the keys.

As shown in Figure 1.a, the buckets formed locally in each processor may vary in size depending on how many keys from each bucket one processor has. We call this variation in size, local data skew. Also, Figure 1.b, shows that the size of a complete bucket may vary depending on the data distribution. We call this variation in size, global data skew (also called “data skew”). Note in Figure 1 that there is local data skew in all processors and a significant global data skew.

As a consequence of local and global skews three problems appear

1. Each key of the data set may have to be moved at most m times among processors producing a locality problem.
2. Global data skew may cause load unbalance.

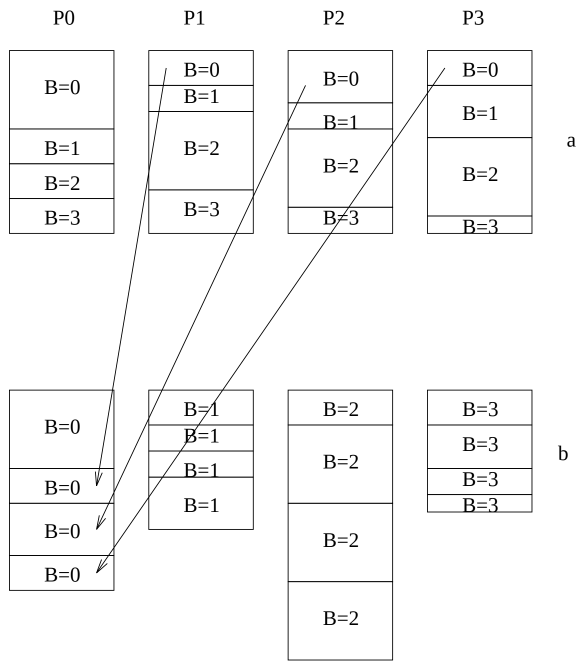

Figure 1: Communication of Parallel Radix Sort. P0 to P3 stand for processor 0 to 3; B=0 to B=3 stand for buckets 0 to 3. (a) Initial state with the same amount of keys per processor. (b) State after one communication state. Each processor has sent buckets B=0 to P0 (shown in figure), buckets B=1 to P1, and so on.

3. Local data skew may cause communication unbalance.

As has been proved in [SoKo98], it is possible to solve the load unbalance problem by sharing buckets among processors. Here, we address the three problems.

3 Communication and Cache Conscius Radix Sort

In this Section, we start by describing the solutions that we propose to increment the data locality of parallel Radix sort and to reduce the communication unbalance. However, the increment of locality rises the problem of load unbalance. We discuss the problem and show how to control it. To end with this section, we describe our proposed algorithm, C^3-Radix sort.

Reducing key movements

To increase the locality of Radix sort, we propose the use of Reverse sorting. This consists in starting the sorting process with the b_{m-1} most significant bits of the key. With this, each processor forms in parallel $2^{b_{m-1}}$ buckets with its local data using the three sequential steps described above (histogram construction, partial sum of counters and data movement). Next, each processor broadcasts the counters so that all processors know the total amount of keys per global bucket. With this knowledge, all processors decide independently what global buckets are assigned to what processors. After that, local buckets are sent to the destination processor to form a global bucket. This is done in an all to all communication.

Note that keys in one bucket have the same value in their b_{m-1} most significant bits. Thus, buckets preserve the final sorting order after a single communication.

Sorting the $b-b_{m-1}$ least significant remaining bits, can be done sepparately for each bucket. At this point, it is possible to use any sequential sorting method given that each bucket is in one processor.

Note that with this strategy, a key only visits two processors at most. Reverse sorting reduces the number of communication steps to only one.

However, the number of reads of the whole set of keys is incremented. The Reverse sorting step performs one read of the whole set of keys to build the histogram of the most significant digit and, one read and one write to move the keys. Then, for each bucket, it is necessary to perform 1 read to build the histogram of the $m-1$ digits that still remain to be sorted and, one read and one write for the movement of each digit. This implies a total of $m+2$ reads and m writes of N keys. This increment in one read of the whole set of keys can be easily compensated by the reduction of the communication to one only step.

Balancing the load

Load unbalance can be detected by all processors once the broadcast of counters is performed. Unbalance happens when it is not possible to distribute full buckets among processors so that an even load is achieved.

We solve the load unbalance problem by applying as many Reverse sorting and broadcast of counters steps as needed, accomplishing what we call Reverse sorting phase. With this strategy, the larger fragmentation of the data set allows more chances to split the buckets for each processor so that the number of keys per processor is balanced.

Balancing communication

One important issue is local data skew after the Reverse sorting phase. We explain how this may influence the amount of data communicated and how this can be dinamically reduced.

Figure 2 shows an example for two processors X and Y that are named Logic processor 0 and 1 respectively. An example for a Reverse sorting phase of one only step with $b_{m-1} = 2$ is given in that Figure. Let us suppose that by the end of the Reverse sorting phase, Logic processor 0 has to store in its memory the set of keys with lower values and Logic processor 1 has to store the set of keys with higher values. Here, we suppose that if one processor has to store more than one bucket, it will store buckets with consecutive values in the b_{m-1} most significant bits, as in our implementation.

In Figure 2.a, we see that Logic processor 0 has 10, 20, 1500 and 4 keys in each of the 4 buckets formed during the Reverse sorting phase. Logic processor 0 is assigned buckets 0 and 1 for the rest of the sorting and Logic processor 1 is assigned buckets 2 and 3. By reasigning Logic processor 0 to processor Y and Logic processor 1 to processor X we only have to communicate 85 keys from processor Y to processor X and 30 keys in the other direction as shown in Figure 2.b. If we had not reasigned Logic processors, we would have ended up having to communicate 1504 and 2002 keys among processors.

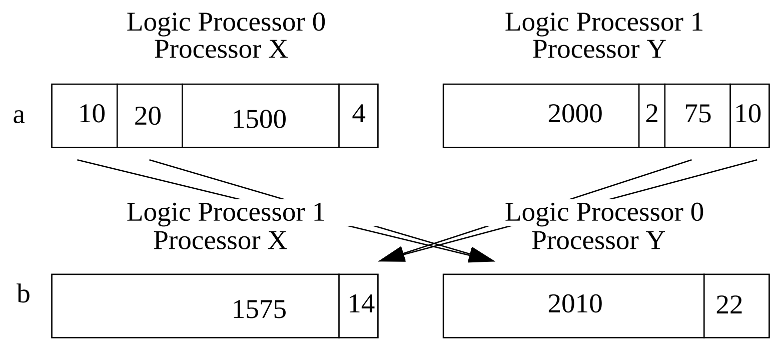

Figure 2: Saving Communication among processors

This rather simple strategy implies little amount of work and does not have any consequence on the way that data are finally sorted if an order vector is kept to determine which real processor holds which Logic processor. Morover, present computers like the SGI O2000 and communication libraries like MPI do not allow the user to choose a specific mapping of Logic processors to real processors. Therefore, this reassignation of Logic to real processors may save a considerable amount of data sent accross the network.

3.1 C^3-Radix sort

The general strategy that we propose is the Communication and Cache Conscious Radix sort, C^3-Radix sort. With this algorithm, the key is divided into two sets of most significant and least significant bits. The most significant bits may be divided into several digits upon which several Reverse sorting steps may be applied depending on the global data skew. The least significant bits are the bucket sorting bits. The method applied to those least significant bits is CC-Radix sort and it is described in detail in [JiNL98], as we said. CC-Radix also applies a Reverse sorting phase to bucket sorting bits to reduce the TLB and cache misses and a Radix sort phase to the remaining bits.

In any case, the size of the digits used for the Reverse sorting and CC-Radix phases are algorithmic parameters and depend on the architecture of the computer. We give details of the size of the Reverese sorting digits in the following paragraphs and in [JiNL98] we give a model to compute the size of the digits for CC-Radix sort.

The description of C^3-Radix sort follows in the next five points

1. Using Reverse sorting, each processor locally builds $2^{b_{m-1}} \geq P$ sub-buckets with the b_{m-1} most significant bits of the key. The choice of b_{m-1} is a trade-off between the number of buckets (it should not be smaller than the number of processors), the number of counters (it should not be too large for communication and memory reasons) and the architectural features of the computer like TLB size, size of the cache hierarchy, etc.

2. The local $2^{b_{m-1}}$ counters are globally broadcasted so that each processor knows the total amount of elements per bucket. If this partitioning achieves good load balance, the algorithm proceeds to point 3. Load balance is achieved if the number of keys in the set of consecutive buckets assigned to any processor (K keys) is within $K < N/P + Thr$, where Thr is the unbalance allowed. Thr depends on the data skew and the architecture of the computer.

   In the case of load unbalance, points 1 and 2 are repeated on the following b_{m-2} bits for all the local buckets. Points 1 and 2 can be repeated as many times as necessary on lesser significant bits.

   Note that each additional Reverse sorting step implies more counters to be broadcasted. In particular, after two Reverse sorting steps, a total of $2^{b_{m-1}+b_{m-2}}$ counters should be broadcasted.

3. In order to reduce communication, all processors decide a reasignation of Logic processors to real processors with the help of the counters received during the last Reverse sorting step.

4. The local buckets of each processor are redistributed as decided in point 3. This is performed with a unique communication of the keys. At the end of point 4, each processor has one or more complete global buckets.

5. The sorting of each global bucket is performed locally in a processor with CC-Radix sort.

4 Experimental setup

The algorithms that we analyze and compare in this paper are tested on the SGI O2000 at CEPBA[1].

The O2000 is a Directory based NUMA Supercomputer [SGI98]. It allows both the use of the shared and distributed memory programming models. However, in order to be able to compare our work to that of [SoKo98] we have used the MPI message passing primitives.

Another important issue is that neither the MPI primitives nor the IRIX 6.4 Operating System allow an explicit control of the mapping of logic processors to real processors.

The building block of the O2000 is the 250 MHz MIPS R10000 processor that has a memory hierarchy with private on-chip 32Kbyte first level data and instruction caches and external 4Mbyte second level combined instruction and data cache. One Node card of the O2000 is formed by 2 processors and 128Mbytes of shared memory. Groups of 2 Node cards are connected to a router that connects to the interconnection network. In our case, the interconnection network is a 2-dimensional hypercube with 4 routers. Therefore, communication between processors that are at different distances varies considerably.

The measures of time have been taken with the help of getrusage() routine and the $libperfex$ library which allows to use the internal hardware counters of the R10000.

The amount of memory per node of the O2000 is 128M bytes. However, we sort a maximum of 16M keys per processor.

Data test sets

The data test sets that we use are 4 of the 5 sets used in [SoKo98]. The sets chosen are Random, Gauss, Bucket and Stagger and we describe them below

- Random forms each 32 bits key calling the C random() routine.
- Gauss adds the results of 4 random() calls and divides this by four.
- Bucket is obtained by setting the first N/P^2 keys at each processor to be random numbers between 0 and $(MAX/P-1)$, the second N/P^2 keys at each processor to be random numbers between MAX/P and $(2MAX/P-1)$, and so forth.

[1]CEPBA stands for "Centre Europeu de Paral.lelisme de Barcelona". More information on CEPBA, its interests and projects can be found at http://www.cepba.upc.es

- Stagger is created as follows. If the processor index is $< P/2$, then all the N/P keys at that processor are set to be random numbers between $(2i+1)MAX/P$ and $((2i+2)MAX/P-1)$, and so forth. Otherwise, the N/P keys are set between $(i+P/2)MAX/P$ and $((i-P/2+1)MAX/P-1)$, and so forth. Here, i ranges from 0 to P.

We decided not to use the Zero data set (all elements set initially to zero) used in [SoKo98] because it would cause no communication to our method and would not be representative.

5 Analysis of C^3-Radix sort

In this section, we analyze the behavior of the Communication and Cache Conscious-Radix sort algorithm on the target computer. First, we analyze how the cycles spent by the algorithm are distributed into Reverse sorting, communication of keys, etc. Second, we study how load and communication unbalance affect the amount of work for C^3-Radix sort.

For the SGI O2000, each Reverse sorting step is performed on 5 bits. The reasons for this stem from the size and use of the TLB. In particular, a user in the SGI O2000 has access to only 48 TLB entries and, therefore, 32 buckets (5 bits) is the number of bits that cause a lower number of TLB misses. More details on this and the relation of the TLB with the memory behavior of the method can be found in [JiNL98].

This value for the number of bits of a Reverse sorting step gives a low number of counters ($b_{m-1} = 32$) which implies a very little overhead due to the broadcast of counters. Therefore, if i Reverse sorting steps are needed due to global data skew, the number of counters to be broadcasted grows to 2^{5i}. In particular, for the data sets used in our experiments we have not needed more than $i = 2$ Reverse sorting steps.

Troughout this and the following sections we give values of Cycles per Sorted Element (CSE) unless stated. For analysis purposes, we use mean CSE, that is, we take the CSE for all the processors involved in one execution and compute their mean. For absolute comparison purposes, we use CSE for the slowest processor. Therefore, the difference between those values (mean and slowest processor) gives an idea of the unbalance between processors.

Cycles distribution

Figure 3 shows CSE distribution for C^3-Radix sort on 16 processors and a randomly distributed data set ranging from 1M to 128M keys. The columns of Figure 3 show the CSE spent for C^3-Radix sort by Reverse sorting, broadcast of counters, computation of bucket distribution, all to all key communication and local sort by CC-Radix. The portions shown in the columns group several steps of Reverse sorting, broadcast, etc. Therefore, they are not in the same order as in the real execution of the algorithm to sepparate communication from computation. Just behind each column in that plot, we show the total slowest processor CSE which determines the execution time of the algorithm.

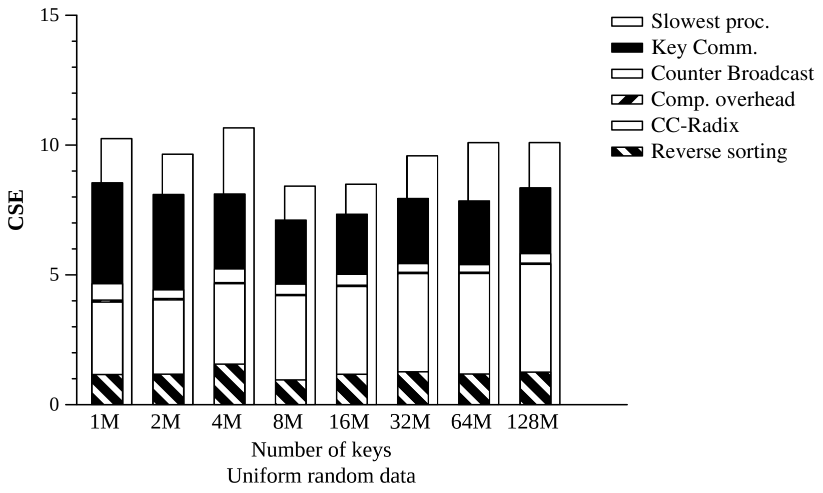

Figure 3: Cycles per Sorted Element for C^3-Radix sort with 16 processors varying the data set from 1M to 128M integers.

In general, we can see that the difference between the mean values of CSE and the slowest processor is about 20%. Also, the slowest processor CSE ranges from 8.5 to almost 11 for different data set sizes. The communication tasks (broadcast of counters and key communication) occupy from a 50% of the total for small data sets to about 30% for larger data sets. The Reverse sorting phase, requires a small amount of the total CSE count, around a 10%. Also, the broadcast of counters is kept to a reduced CSE count.

Load and communication balance

To analyze the load and communication balance of C^3-Radix sort we have used different skewed data sets. Those are Random (R), Gaussian (G), Bucket (B) and Stagger (S) data as described in Section 4.

Let us see how the algorithm parameter Thr (amount of load unbalance allowed per processor) influences the total execution time. Figure 4 shows CSE as a function of different data set sizes for 16 processors and the Gaussian distribution of keys. The plot shows the values for unbalance thresholds ranging from $Thr = 32K$ keys through $Thr = 512K$ keys. We use a Gaussian data distribution because it has large data skew.

We can say that appart from the odd behavior of $Thr = 32K$, the rest of allowance thresholds behave with very little differences. Therefore, given the little influence of this parameter we choose a rather large value of $Thr = 128K$ keys for our measures.

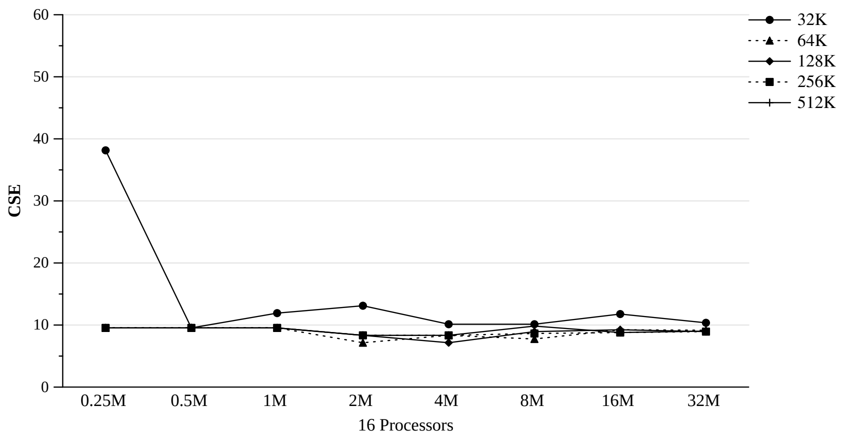

Figure 4: CSE caused by different values of the Thr parameter in C^3-Radix sort.

Figure 5 shows results for 16 processors varying the data set from 2M to 128M keys. It is noticeable that Gaussian distribution requires more Reverse sorting cycles than the other distributions due to load unbalance. This is because a second Reverse sorting step will be necessary in order to have small data subsets per bucket and balance the load for the local sort. However, note that the total amount of Reverse sorting plus CC-Radix sort work does not increase considerably for Gaussian compared to the rest of methods. The reason is that each Reverse sorting step already sorts some bits of the key which saves work to the local CC-Radix sort.

The effect of communication unbalance can be noticed in Figure 5, too. There, Stagger data sets have a significant amount of work devoted to the computation of bucket distribution. Stagger is a specially ill posed case for communication because each Logic processor has data that will be eventually assigned to other Logic processors.

As a conclusion for this section we can see in Figure 6 that C^3-Radix sort is insensitive to data skew. In this plot, we show the slowest processor CSE for different numbers of processors keeping the key set fixed to 16M keys.

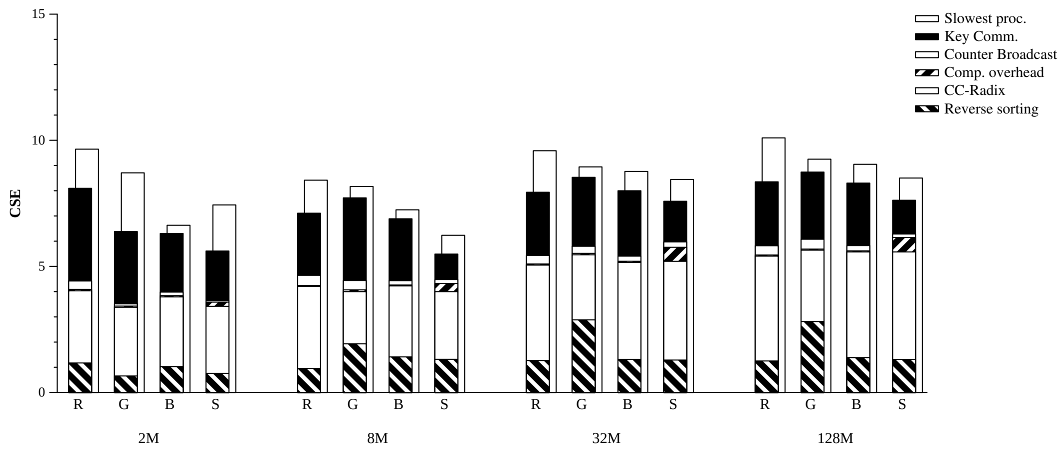

Figure 5: Comparison of four different data distributions for C^3-Radix sort. Data sets vary from 2M keys to 128M keys for 16 processors. (R) Random, (G) Gaussian, (B) Bucket and (S) Stagger distributions.

6 Previous work

Previous work in the area of Distributed Memory Parallel sorting has been directed to implement versions of Radix sort that improve communication [Hel96] and load balance [SoKo98].

In [Hel96], the authors propose the use of an algorithm that they call Personalized Communication for different Parallel sorting algorithms. The idea behind Personalized Communication is to reduce the number of communication lost cycles due to processor skew. They give results for Random sample sorting, Regular sample sorting and two versions of Radix sort. They show that their communication strategy reduces the amount of lost cycles and balances the communication no matter what data distribution they have.

In [SoKo98], the authors implement a parallel version of the basic version of Radix sort that aims at balancing the computation. They call their algorithm Load Balanced Radix sort. The algorithm implements an enhancement that improves the load balance among processors before sorting digit. Their approach is to broadcast the counters right before communicating the data to decide how many keys from each bucket must travel to which processor. Their algorithm achieves a really good balance of work among processors. The authors say that their strategy does too many communication steps and that this should be investigated. The authors show that their algorithm is 30% to 100% times faster than those analyzed in [Hel96].

The differences between those works and ours is that our aim is to reduce

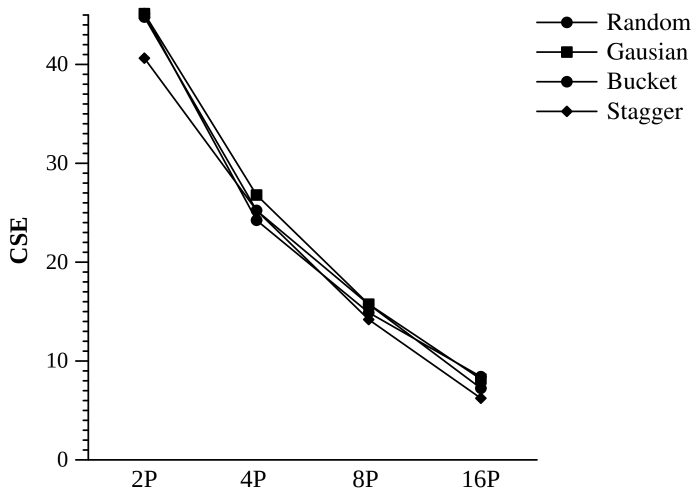

Figure 6: Comparison of four different data distributions for C^3-Radix sort. The number of processors grows from 2 to 16 keeping the amount of keys constant to 16M.

the number of communication steps and to reduce the total amount of data communicated achieving a good load and communication balance while their objectives are as explained before.

Finally, we must mention that the idea of partitioning the key set comes from the use of Bucket sort [Knu73] and was used by Agarwal in [Aga96]. Our proposal differs from [Aga96] in the fact that we use Radix sort instead of Bucket sort to perform the partitioning. In some previous work [Lar97], we showed that Radix sort requires less memory and is faster than Bucket sort in order to do the partitioning of data sets.

7 Comparison with previous work

In this section we compare C^3-Radix sort with Load Balanced Radix [SoKo98] which outperformed other competitive methods and showed to be up to 2 times faster than those methods sometimes. The comparison in [SoKo98] showed figures for other Radix sort implementations [Ale95, Hel96] and Sample sorting algorithms described in [Hel96].

Therefore, given that Load Balanced Radix is the fastest present method to our knowledge we want to understand the differences of that method with the C^3-Radix sort that we propose here.

Figure 7 shows a comparison of the CSE varying the number of processors and keeping the number of keys constant to 16M for Load Balanced Radix and C^3-Radix sort. The plots show that C^3-Radix sort is more than three times faster than Load Balanced Radix for a small number of processors and about three times faster for a large number of processors.

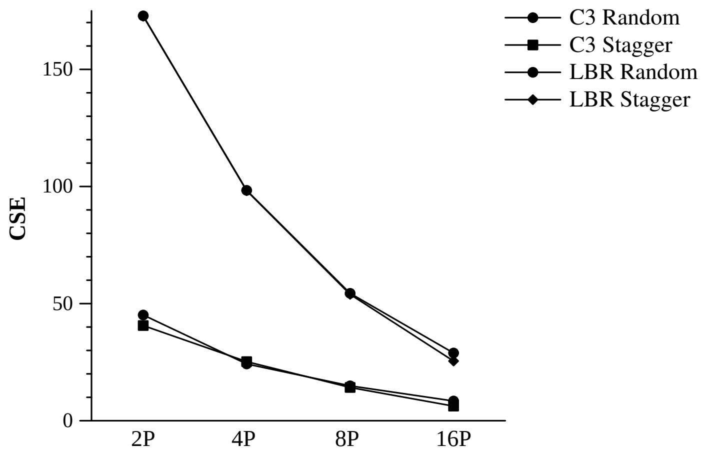

Figure 7: CSE caused by different data distributions on C^3-Radix sort and Load Balanced Radix.

Now, we turn to analyze the differences between the two methods. Figure 8 shows a comparison of both Load Balanced Radix and C^3-Radix sort for 16 processors varying the amount of data. The bar diagrams show how execution cycles are spent by the methods for Random, Gaussian and Stagger data distributions. We do not show the Bucket distribution results because they are quite similar to the Random distribution results.

Communication is more than three times better for C^3-Radix sort than for Load Balanced Radix. This is because Load Balanced Radix performs four communication and broadcast steps while C^3-Radix sort only performs one communication and one or more broadcast steps. The number of broadcast steps depends on the number of Reverse sorting steps performed. In addition, C^3-Radix sort reduces the amount of data communicated while Load Balanced Radix does not do so.

Load balance is treated differently by both C^3-Radix sort and Load Balanced

Radix. The former gives some unbalance allowance to have full buckets in each processor. The latter performs an even distribution of keys among processors without caring about having full buckets in each processor. However, Figure 8 shows that the difference between the mean execution time and the slowest processor is not so significant for both methods. Moreover, the Random case has a larger deviation for both methods. Also, for small data sets, C^3-Radix shows a larger deviation. This is because the unbalance allowance is proportionally large for such data sets.

Finally, we want to understand the differences of the local sorting execution CSE. By local sorting we understand the work undertaken by the processor and memory hierarchy that does not correspond to communication. The local Radix sort CSE of Load Balanced Radix are variable for different data set sizes. On the other hand, the CC-Radix sort CSE of C^3-Radix sort are constant. This is because C^3-Radix sort exploits the memory hierarchy better than Load Balanced Radix. While C^3-Radix sort partitions the data set with the Reverse sorting phase, Load Balanced Radix does not partition the data set and traverses all the data assigned per processor for each digits. In addition, C^3-Radix sort makes use of CC-Radix sort as a sequential algorithm for the local sort which exploits the memory hierarchy better than plain Radix sort [JiNL98]. For the case of Load Balanced Radix, memory usage has two clear behaviors, one for data sets that fit in the L2 cache and do not cause TLB interferences and, another one for large data sets that cause such TLB interferences.

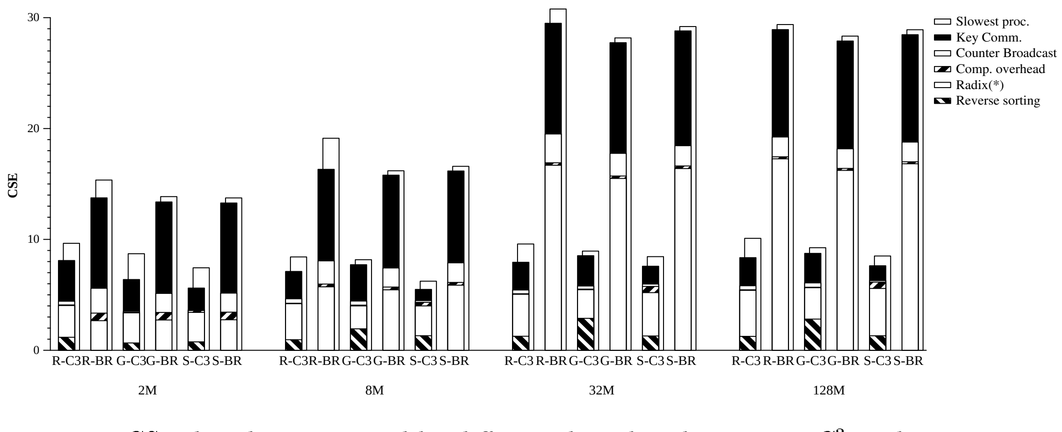

Figure 8: CSE distribution caused by different data distributions on C^3-Radix sort and Load Balanced Radix. (*) The portion of bar corresponding to Radix sort stands for CC-Radix in the case of C^3-Radix and plain Radix for Load Balanced Radix. (R) Random, (G) Gaussian and (S) Stagger distributions.

Finally, Figure 9 shows a comparative plot of speed-ups for both C^3-Radix sort and Load Balanced Radix. The clear difference in speed up shows that

C^3-Radix sort is a much adaptive version than Load Balanced Radix.

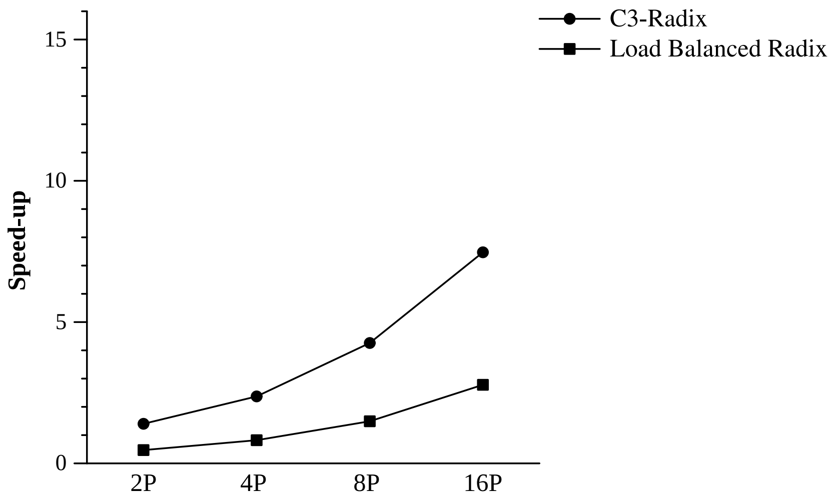

Figure 9: Speed-up comparison between C^3-Radix sort and Load Balanced Radix.

8 Conclusions

In this paper we have proposed a variation of the parallel Radix sort algorithm that we call Communication and Cache Conscious Radix sort, C^3-Radix sort.

Our algorithm has three important achievements. First, it increments the data locality by using a technique that we call Reverse sorting. With this technique, we reduce the number of communication steps of the original Radix sort [SoKo98] from 4 to only 1. Second, we reduce the amount of data communicated among processors during the single communication step of the algorithm. We do that by reasigning Logic processors to real processors instead of moving data. Third, we achieve a reasonable load and communication balance. Those three important achievements lead to a method that is invariant to skewed data distributions.

C^3-Radix sort outperforms previous work presented in [SoKo98] which was the fastest parallel sorting algorithm to our knowledge. Our results show that our algorithm is more than 3 times faster than the previous fastest sorting algorithm for data sets larger than 8M keys on 16 processors and 2 times faster for smaller data sets. Also, our communication is equivalent to 1/3 of that of Load Balanced Radix.

C^3-Radix sort shows a good speed-up of 7.3 for 16 processors and 16M keys while previous work only achieves a speed-up of 2.5 for the same set up.

References

[Aga96] R. Agarwal, A Super Scalar Sort Algorithm for RISC Processors, IBM Research Report, January 1996.

[Ale95] A. Alexandrov, M. Ionescu, K. Schauser and C. Scheiman, LogGP: Incorporating long messages into the LogP Model- One step closer towards a realistic model for parallel computation. In Proc. of the ACM Symposium on Parallel Algorithms and Architectures, pp. 95-105, Santa Barbara, CA, July 1995.

[Arp97] A. Arpaci-Dusseau, R. Arpaci-Dusseau, D. Culler, J. Hellerstein and D. Patterson, High-Performance Sorting on Networks of Workstations, SIGMOD'97 Conference, pp. 243-254.

[DNS92] D. DeWitt, J. Naughton, D. Schneider, Parallel Sorting on Shared Nothing Architecture Using Probabilistic Splitting, Proc. of the First Intl. Conf. on Parallel and Distributed Info Systems, IEEE Press, pp. 280-291, 1992.

[FrM88] R. Francis and I. Mathieson, A Benchmark Parallel Sort for Shared Memory Multiprocessors, IEEE Transactions on Computers, Vol. 37, No. 12, Dec 1988.

[Hel96] D. Helman, D. Bader and J. JaJa, Parallel Algorithms for Personalized Communication and Sorting with an Experimental Study, IEEE Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 211-220, Padua, Italy, 1996.

[JiNL98] D. Jimenez-Gonzalez, J. Navarro, J.-L. Larriba-Pey, Cache Conscious Radix Sort, Research Report DAC-UPC 70-98. Submitted.

[Knu73] D. Knuth, The art of Computer Programming; Volume 3/Sorting and Searching, Addison-Wesley Publishing Company, 1973.

[Lar97] J. Larriba-Pey, D. Jimenez and J. Navarro, An Analysis of Superscalar Sorting Algorithms on an R8000 Processor, Proceedings of the Intl. Conf. of the Chilean Computing Society, Valparaiso, Chile, Nov. 1997. pp. 125-134, published bythe IEEE society.

[Nyb94] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray and D. Lomet. AlphaSort: A RISC Machine Sort. Proceedings of the Sigmod 94 Conference, pp. 233-242.

[SGI98] R. van der Pas, SGI Porting and Optimization Seminar, Course given at CEPBA.

[SoKo98] A. Sohn, Y. Kodama, Load Balanced Parallel Radix Sort, International Conference on Supercomputing, Melbourne, Australia, 1998.