

Sorting on the SGI Origin 2000: Comparing MPI and Shared Memory Implementations *

D. Jiménez-González

E. Guinovart

J.-L. Larriba-Pey

J. J. Navarro

Computer Architecture Dept.

Universitat Politècnica de Catalunya

Jordi Girona 1-3, Campus Nord-UPC, Modul D6, E-08034 Barcelona

e-mail: {djimenez,enricg,larri,juanjo}@ac.upc.es

Abstract

In this paper we analyse the Communication and Cache Conscious Radix sort algorithm, C^3 -Radix, using the distributed and the shared memory parallel programming models.

C^3 -Radix was originally proposed based on the idea of the classic Radix sort to exploit the memory hierarchy locality and reduce the amount of communication for distributed memory computers. Here, we implement C^3 -Radix on the SGI Origin 2000 NUMA multiprocessor and make use of the Message Passing Interface (MPI) and the native shared memory directives of that computer to implement the two programming models that we want to analyse.

We give results for up to 16 processors and 64M 32bit keys. The results show that for data sets that are small compared to the number of processors, the MPI implementation is faster while for data sets that are large, the shared memory implementation is faster. In the paper, we explain the reasons for the different behaviours depending on the size of the data sets.

1. Introduction

The Communication and Cache Conscious Radix sort algorithm (C^3 -Radix) was proposed as a message passing alternative to the fastest sorting algorithms proposed in the literature [7]. The algorithm was programmed using MPI communication primitives.

*This work was supported by the Ministry of Education and Science of Spain under contract TIC-0511/98 and by CEPBA. Daniel Jiménez-González is supported by "Direcció General de Recerca of the Generalitat de Catalunya" under grant 1998FI-00283-APTIND.

C^3 -Radix improves the data locality at the memory hierarchy level and reduces the communication steps to only one, assuring load balance among processors.

C^3 -Radix makes use of CC-Radix sort which is a sequential version of Radix sort that reduces cache and TLB misses. This sequential version of Radix sort is analyzed in detail in [8].

Here, we present the shared memory version of C^3 -Radix and compare its behaviour to the distributed memory implementation on the Silicon Graphics Origin 2000 (SGI O2000).

The SGI O2000 is a Non Uniform Memory Access architecture (NUMA) with physically distributed memory and logically shared memory [13]. Within the SGI O2000, every two processors have their own shared memory and can communicate with the rest of processors through a hypercube interconnection network. However, a directory structure allows to have a unique virtual address space and multiprocessor cache coherence. With this, processors can access data in their memory (local) as well as data in the memory of other processors (remote) by simply using a memory address. The only noticeable difference between local and remote accesses is their latency which may be from 3 to 4 times larger for remote data than for local data.

The SGI O2000 can be programmed using both the distributed or shared memory programming models. On the distributed side, one can use MPI or PVM message passing libraries. Here, we have used MPI.

On the shared side, the SGI O2000 offers the possibility to use a native shared memory model. In this case, the structure of a NUMA machine is transparent to the High Level Language programmer for whom it is easier to program. However, when a program is not designed to take into account the structure of the computer, it may suffer from large memory delays to access remote memory positions.

A good allocation of data can be achieved with the directives provided by the compiler. With those directives, the programmer can tell the compiler where to place data in the proper memory area to make sure that each processor will have its data set in its local memory.

The two important outcomes of the paper are as follows. First, the results that show that for small data sets, i.e. 1M keys, the MPI implementation is faster while for large data sets, i.e. 64M keys, the shared memory implementation is faster. Second, the analysis of how the different steps of the algorithm are executed using both programming models.

The paper is structured as follows. In section 2 we give an account of the previous work in the topic of sorting. In section 3 we explain the basics of the original sequential and parallel versions of the Radix sort algorithm for completeness. In Section 4 we explain C^3 -Radix sort in its distributed and shared memory model versions. In Section 5 we describe the experimental setup that we utilize. Section 6 is devoted to compare the execution of the two versions of the algorithm and understand their behaviour on the SGI O2000. Finally, in Section 7 we conclude.

2. Previous work

In the field of parallel sorting, efforts have been addressed to solve the problem for disk resident data [4, 11, 1, 3] and for memory resident data [5, 6, 12].

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 the disk resident papers [11, 1, 3] and one of the memory resident papers [6], some memory hierarchy locality issues are addressed. However, to our knowledge, a comparison of the behaviour of shared and distributed programming models and the way they exploit locality has not been addressed explicitly yet for Radix sort.

3. Classic Sequential and Parallel Radix sort

Let us suppose that we want to sort N integer keys of b bits each. Radix sort is based on the idea of sorting the key by 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 makes use of any other sorting method. In our case, we use the counting algorithm [9] which is divided into three steps. These steps can be followed with Figure 1 where an example of 2 decimal digit keys is given. First, using the values of digit i , the method builds a histogram of the N keys on 2^{b_i} counters (count). Second, 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 (accum.). Third, the keys are moved to the corresponding buckets in the destination vector with the help of the indices created previously. Note that the counting algorithm requires auxiliary memory of size the set of keys to be sorted, to hold the destination vector. Also, it requires additional space for the set of counters. This sequential algorithm is described in [9].

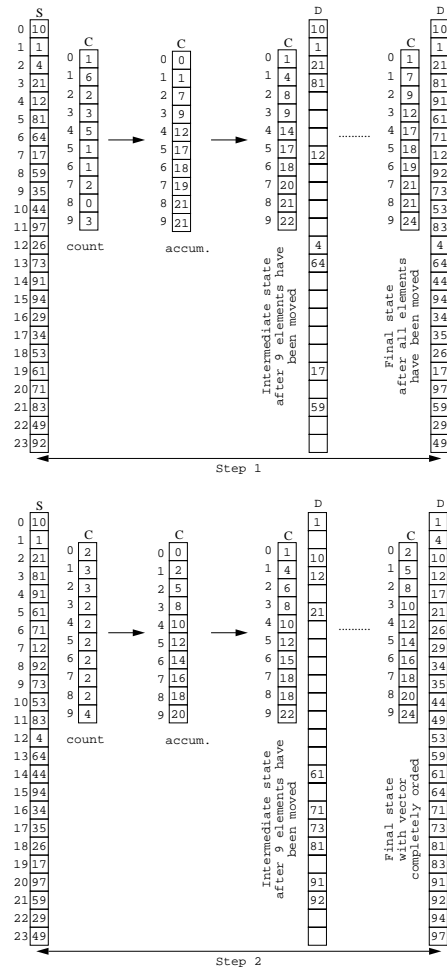


Figure 1. Radix sort for 2 decimal digit keys.

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 across 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} local buckets are formed in each processor. Note that the data contained in each local bucket is a subset of the data of a global bucket. Now, the objective is to perform an all to all communication so that full global buckets are placed in the same processor. After

that, the sorting of the b_1 bits of the second least significant digit starts. This will be repeated for each of the digits of the keys. Depending on the data distribution, the sorting of the rest of digits will lead to considerable load unbalance given that Radix sort requires each global bucket to be placed in the memory of one only processor to avoid extra communication. The problem of Load Unbalance for the classic parallel version of Radix sort was addressed in [12]. There, global buckets could be shared by as many processors as necessary to avoid unbalance.

The shared memory version of the classic Radix sort differs from the distributed memory one in the fact that each communication step is substituted by a synchronization.

An important aspect of the distributed memory version of Radix sort is that it requires a communication step after the local sorting of each digit. In the case of the shared memory version, reads and writes are made to remote memory incurring a lot of extra stall time. This, together with the problem of load balance, is solved with the Communication and Cache Conscious Radix sort that we explain in the following section.

4. C^3 -Radix Sort

To increase the data locality of Radix sort on distributed memory computers, we proposed how to make use of Reverse sorting [7]. This consists in starting the sorting process with the b_{m-1} most significant bits of the key on the local data. With this, each processor forms in parallel $2^{b_{m-1}}$ local 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. In our implementation, we assign full global buckets to each processor. After that, local buckets are sent to the destination processor to form a global bucket. This is done in an all to all communication phase.

For the case of shared memory computers, the Reverse Sorting phase is performed similarly and each processor forms in parallel $2^{b_{m-1}}$ local buckets using a subset of all the keys. This subset does not intersect with the keys accessed by other processors. Each processor also makes use of private counters.

After the Reverse Sorting phase, a global synchronization is executed and all the processors read the counters of the rest of processors. Once it is decided what global buckets are assigned to what processors, the data movement is performed so that data belonging to one global bucket are moved to contiguous memory positions. Then, each processor performs the local sort on the global buckets assigned to it.

Some comments can be made to Reverse Sorting, how it is implemented and the effects that it has on the sorting of the least significant digits:

- After the reverse sorting, keys in one bucket have the same value in their b_{m-1} most significant bits. Thus, global buckets get sorted among them after that phase.
- Sorting the $b - b_{m-1}$ least significant remaining bits, can be done separately for each bucket. At this point, it is possible to use any sequential sorting method given that each bucket is assigned to one processor. In particular we use CC-Radix sort [8].
- In order to balance the load of processors, we perform as many Reverse Sorting steps as necessary. With each of those steps, the set of buckets increases and the granularity of those buckets decreases. This gives more chances of having smaller buckets which may lead to a balance in the load.
- Finally, we reduce the amount of data communicated by reassigning logic processors to physical processors dynamically [7]. With this, we reduce the global amount of data communicated and avoid unnecessary communication.

4.1. Implementation of C^3 -Radix

We can think of C^3 -Radix sort as if the key was 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. This number of bits can vary and depends 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 [8]. CC-Radix also applies a Reverse sorting phase to the 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 Reverse sorting digits in the following paragraphs and in [8] we give a model to compute the size of the digits for CC-Radix sort.

The description of message passing C^3 -Radix sort follows in the next five points. After that description, we comment on the differences with the shared memory version of the algorithm.

1. *Reverse sorting.* Using Reverse sorting, each processor builds $2^{b_{m-1}} \geq P$ sub-buckets locally 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. *Broadcast of counters.* 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 $K < N/P + Unb$, where Unb is the unbalance allowed. Unb 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. *Computation of bucket distribution.* In order to reduce communication, all processors decide a reassignment of Logic processors to Real processors with the help of the counters received during the last *Broadcast of counters* step.
4. *All to all key communication.* 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. *Local sorting by CC-Radix sort.* Each processor sorts the global buckets assigned to it with CC-Radix sort.

The main differences between the Distributed and Shared Memory implementations of the algorithm are as follows:

- The key vector is declared as a set of local independent subvectors in the MPI version and as a vector with a unique address space for the Shared Memory version. However, in the shared memory case, we make use of compilation directive `page_place` that helps to assign chunks of the data vector to the memory of a specific processor.
- The *Broadcast of counters* phase is changed into a global synchronization and a set of remote accesses of each processor to the counters of the rest of processors.

- *Computation of Bucket Distribution* is implemented so that the number of remote accesses is reduced in the phases that follow.
- The *All to all key communication* is translated into remote accesses to data.

5. Experimental setup

The algorithms that we analyze and compare in this paper are tested on the SGI O2000 at CEPBA¹.

The building block of the SGI 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 data and instruction cache. One Node card of the SGI 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.

One 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.

In the shared memory implementation of our algorithm, we have made use of directive `page_place` that allows the programmer to place variables in the local memory of specific logic processors. This way, we are able to control where data are placed and exploit data locality as much as possible.

The measures of time have been taken with the help of the `getrusage()` routine.

The measures given in the following section are for Randomly distributed data. In this paper we have not shown results for other distributions because we showed that our algorithm is robust in such cases in [7].

6. Analysis of C^3 -Radix sort

In this section, we analyze the behavior of the shared memory version of Communication and Cache Conscious-Radix sort algorithm on the SGI O2000 and compare it to the distributed memory one. First, we make a global comparison of the shared memory version and the distributed memory version of the algorithm. Second, we analyze how the cycles spent by the algorithm are distributed into the different phases of the algorithm.

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 of the SGI O2000

¹CEPBA 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>

has access to only 48 TLB entries and, therefore, 32 buckets (5 bits) is the number of bits that cause a smaller 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 [8].

Troughout this section we give values of Cycles per Sorted Element (CSE). CSE is the number of machine cycles taken to execute the algorithm divided by the number of elements to be sorted. We have used the number of cycles taken by the slowest processor in order to show execution speed. Moreover, we have used the mean number of cycles taken by all the processors involved in the computation in order to analyze the trends for the different phases of the algorithms.

Figure 2 shows the CSE for the slowest processor for each of the executions of the two implementations of C^3 -Radix sort. The plot gives results for 16 processors and 1M to 64M 32 bit keys. The main difference between both implementations is how they behave for small and large data sets. For sets of keys up to 8M keys, the MPI implementation is faster than the shared memory one. For larger data sets, the shared memory version is faster. Now, we concentrate on the details of both implementations to understand their different behaviour.

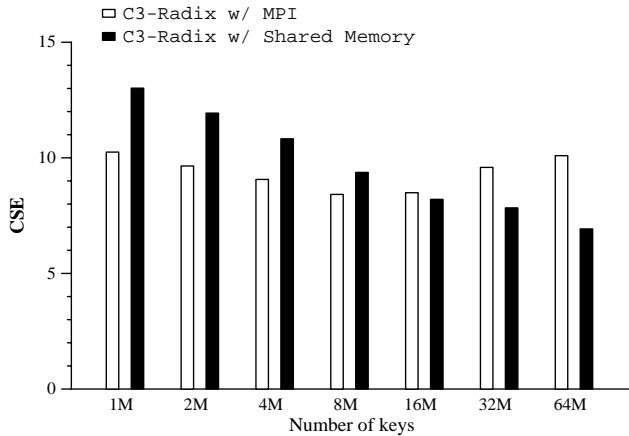


Figure 2. Cycles per Sorted Element for the slowest processor of each execution of the MPI and Shared Memory versions of C^3 -Radix sort with 16 processors varying the data set from 1M to 64M integers.

Figure 3 shows the CSE for 16 processors and 1M to 64M 32bit keys for both the distributed (1st column of each pair) and shared memory versions (2nd column of each pair) of the algorithm. The results shown in the plot stand for the mean CSE of all the 16 processors involved in one execution. There, we can see the different phases of the algorithms. Equivalent phases in both implementations are

named the same although they are implemented differently. For instance, the *Counter Movement* phase performs explicit communication in the MPI implementation while it is implemented with a global synchronization and remote accesses in the shared memory version.

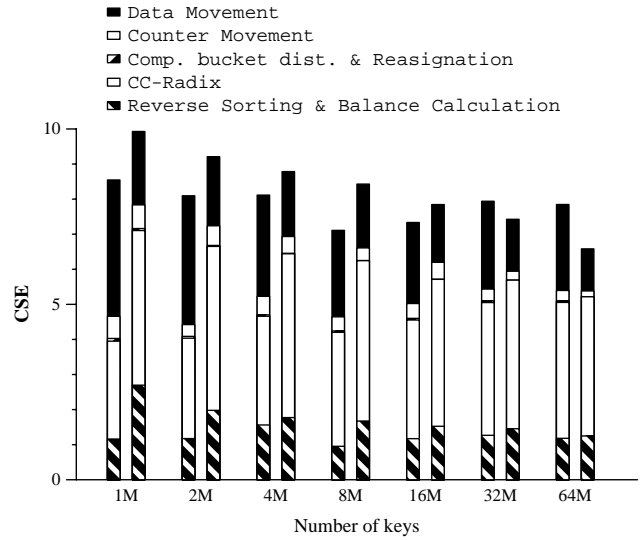


Figure 3. Distribution of Cycles per Sorted Element for the different stages of the MPI and Shared Memory versions of C^3 -Radix sort with 16 processors varying the data set from 1M to 64M integers. The first column of each pair stands for the MPI implementation and the second column stands for the shared memory implementation. The results shown in the plot stand for the mean CSE of all the 16 processors involved in one execution.

Now, we comment on the differences for each of the phases

- *Reverse Sorting & Balance Calculation* phases take more cycles for small sets of keys in the shared memory version than in the distributed memory version. The reason for that is twofold. First, it is caused by the mapping of data that forces directive `page_place` in the shared memory version. As a matter of fact, for small data sets this directive does not place data as expected, that is, some data are placed remotely. As a consequence, most of the accesses that should be made to local memory are made to remote memory. Second, if data are stored in the memory of a few processors there may be memory contention due to multiple accesses concurrently by remote processors. Those two aspects increase the amount of CSE for the shared memory version considerably. Those problems do not appear in the MPI implementation because this phase is performed on local data.

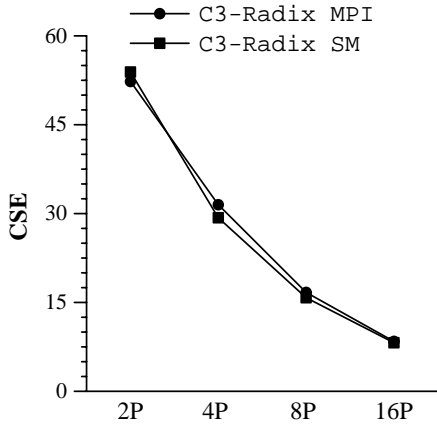


Figure 4. Comparison of the MPI and Shared Memory versions of C^3 -Radix sort varying the number of processors from 2 to 16 with a data set fixed to 16M keys.

- *CC-Radix* phase is influenced by the same aspects as the *Reverse Sorting & Balance Calculation* phases.
- The amount of CSE for both the *Computation of Bucket distribution and Reassignment of processors* and *Counter movement* is similar for both implementations.
- *Data Movement* has a larger CSE for the distributed memory implementation. There are two reasons for that. First, the MPI primitives make use of memory buffers for communication. Therefore, there may be double copy of data at each end of the communication. Second, there is an explicit synchronized communication between the sender and receiver processors which may introduce synchronization delays. Only one processor receives data from a specific processor at a specific moment.

On the other hand, the shared memory implementation allows all processors to be reading from any memory position at the same time. This allows overlapping of reads and writes due to the fine granularity of such accesses. In other words, the shared memory implementation performs asynchronous access to data reducing the amount of stall cycles in that phase.

Figure 4 shows a comparison of the two implementations for a fixed amount of keys (16M keys) on a varying number of processors. There, we can see that there is not a significant difference between the implementations when the

number of processors varies and the amount of data is kept constant.

Finally, we want to understand how efficient are the algorithms and we measure that by means of the speed-up. Figure 5 shows the speed-up of the distributed memory version of C^3 -Radix sort. We only show this version of the algorithm because both implementations behave similarly for 16M keys as we showed above. We measure the speed-up against the best sequential algorithm the we know, *CC-Radix* sort. We can see there that a speedup of around 8 is achieved for the algorithm on 16 processors. This is quite remarkable given that speed-ups of only 3 were achieved with the previous fastest sorting algorithms [7].

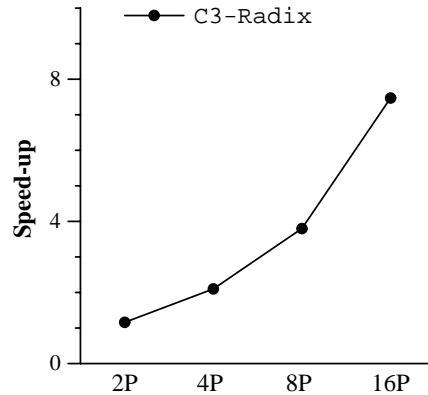


Figure 5. Speed-up of C^3 -Radix sort.

7. Conclusions

In this paper we have compared a distributed and a shared memory implementations of the C^3 -Radix sort algorithm on the NUMA computer SGI O2000.

In general, the shared memory version of the algorithm behaves better for large data sets while the distributed memory version is better on small data sets.

The reasons for this different behaviour stem from the characteristics of the data movement between processors and data placement of both models

- Communication is explicit with the distributed version of the algorithm. In this case, there is a need for buffering and synchronized communication which increments the amount of communication time. On the other hand, shared memory allows concurrent memory accesses by different processors. This reduces the communication time.

- The data placement in the shared memory implementation causes remote accesses during phases that are performed on local data in the MPI version. This is especially noticeable for small data sets where the total amount of incremental time due to the placement overweighs the gain due to the characteristics of communication explained in the previous point.

References

- [1] R. Agarwal. A super scalar sort algorithm for risc processors. *IBM Research Report*, January 1996.
- [2] A. Alexandrov, M. Ionescu, K. Schausser, 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*, pages 95–105, July 1995.
- [3] A. Arpaci-Dusseau, R. Arpaci-Dusseau, D. Culler, J. Hellerstein, and D. Patterson. High-performance sorting on networks of workstations. *SIGMOD Conference*, pages 243–254, 1997.
- [4] D. DeWitt, J. Naughton, and 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, pages 280–291, 1992.
- [5] R. Francis and I. Mathieson. A benchmark parallel sort for shared memory multiprocessors. *IEEE Transactions on Computers*, 37(12), December 1988.
- [6] 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*, pages 211–220, 1996.
- [7] D. Jimenez-Gonzalez, J.-L. Larriba-Pey, and J. Navarro. Communication conscious radix sort. *Proceedings of Intl. Conference on Supercomputing*, ACM Press, pages 76–82, June 1999.
- [8] D. Jimenez-Gonzalez, J. Navarro, and J.-L. Larriba-Pey. Cache conscious radix sort. *Research Report (Submitted)*, (DAC-UPC 70-98), 1998.
- [9] D. Knuth. *The art of Computer Programming; Volume 3/Sorting and Searching*. Addison-Wesley Publishing Company, 1973.
- [10] 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*, pages 125–134, November 1997.
- [11] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: A risc machine sort. *Proceedings of the Sigmod Conference*, pages 233–242, 1994.
- [12] A. Sohn and Y. Kodama. Load balanced parallel radix sort. *International Conference on Supercomputing*, 1998.
- [13] R. van der Pas. Sgi porting and optimization seminar. *Course given at CEPBA*.