Sorting on the Cray X1

Helene E. Kulsrud
CCR-P

Introduction

Since 1978 and our studies of the Cray I, CCR has used sorting of 63 and 64 bit numbers as a benchmark. Therefore it was expected that we would be revisiting our sorting programs when we received our new Cray X1. In an effort to help Cray, Inc. produce the necessary software for the new architecture, we agreed to provide sorting programs for our site and for the Cray library. This seemed like any easy project since we had revised our CCR software libraries to be portable and had ported them to various other architectures such as T3E, Alphas, Sparcs and Origins. Each porting has required changes due to assumptions about word sizes (particularly for C programs), retired language constructs and increasing strictness of current compilers. We were prepared for some minor changes.

We can report that compiling and execution of the portable sorting programs was successful and did require a few changes. But the resultant execution times for the programs were not satisfactory. We therefore embarked on a project to improve the execution times. In this paper, we will show how codes were improved, We will also suggest further improvements both to our own codes and to Cray compilers.

Sorting Algorithms

There are many popular sorting methods such as:

Heap Sort
Bubble Sort
Insertion Sort

Shell Sort
Mergesort
Quicksort
Radix Sort
Radix Exchange Sort
etc……
Parallel Sorts etc., etc…..

In our experience and in the literature[1], however, only three sorts are useful for large data sets. What is meant by large is vague but certainly 500 words or elements is meant. In this paper, three of the many sorts in our library, will be presented. Each sort illustrates different aspects of the machine. These are:

QKSORT - A scalar sort(radix exchange)
PSORT - A vectorized sort(radix)
MSORT - A multiprocessor sort

QKSORT

Radix Exchange sort is a variation on the standard popular Quicksort. In Quicksort, a pivot element is chosen. There are various methods suggested for choosing the pivot. The data is then divided into two parts one of which contains all elements which are less than or equal to the pivot and the other part which contains all elements which are greater than the pivot. Starting from the first and last elements in the array, elements which belong to the less than/equal group are interchanged with elements which belong to the greater than group. This is done one element at a time. When the exchange is completed there are two data groups which can then be sorted separately.
This is a recursive process as each group is divided in two until only one word remains in each group. In Radix Exchange sort, the two groups are created by finding the leftmost bit that contains both zeros and ones and then using the single bits as the pivot elements.

It is natural to consider that when a set of numbers is rearranged then the two sub sorts could be carried out as separate computer processes. That is, after the data is separated, a process can be spawned so that two, then four, etc. sorts occur simultaneously. We have programmed this in the past and find that the cost of spawning processes requires fairly large data sets. (i.e. 1 million words). This is due to the high cost of process creation.

Our program to sort using radix exchange is called QKSORT. It consists of two basic parts. The first part which carries out the interchange of the elements, and the second part which does an insertion sort for sets which contain a number of elements less and or equal to L, the number of elements in a vector register (64 or 128). To use the power of a vector machine, we turn these scalar algorithms into vector algorithms.

The basic idea for a vector version of the interchange is to load one vector register with the first group of L elements and another vector register with the last group of L elements. We then use mask instructions to find which elements do not belong in the groups, compress these elements into a group and create two new vectors which have only the correct elements. The two loaded vectors will be used up at different rates which results in several branches in the program. We must also load vectors before they are needed so as not to delay the computation.

The basic idea for the short sort is to load a vector register with the L or less elements for an insertion sort. We then compare each element, E against all the other elements in the list to compute how many other words are less than or equal to E, say M, and place E at the Mth position in another vector register . In order to deal with ties, 1 is then added to E. When all the elements have been processed, the new vector is complete and is stored. Since current compilers cannot produce this kind of code, the two parts are written in assembly language.

Table I gives the calculated times for the ported version of QKSORT and the new version with the assembly language

programs. All times in this paper are presented in seconds. The tests in Table I were run for 53 bit numbers randomly generated numbers. These programs use only one SSP.

Table I
Improving QKSORT

Count	5,000	10,000	50,000	100,000	500,000	1,000,000	4,000,000
Version							
Port	.016	.037	.217	.467	2.784	5.987	27.208
New	.002	.003	.018	. 038	.205	.425	1.824

Table 2 compares QKSORT on the X1 with some other machines for randomly generated 63 bit numbers. The Alpha, an outstanding scalar machines produces the best code. The Cray X1 runs in about half the T90 time and should be able to do better.

Table 2
QKSORT in 2003

Size Machine		10,000	50,000	100,000	500,000	1,000,000	5,000,000	10,000,000
T3E		.006	.037	.079	. 510	1.024		
T90		.009	.040	.081	.411	.831	4.254	8.364
Alpha EV6.7		.002	.017	.035	.208	.453	2.702	5.751
Alpha EV6.8		**.001**	**.007**	**.014**	**.082**	**.175**	**1.200**	**2.720**
X1		.004	.019	.039	. 215	.449	2.426	5.008

PSORT

PSORT is a program which implements the radix sort method. The idea here is to choose a pocket of Q bits and count how many of the elements have each of the values the pocket will have. We start with the right most bits in the word. Given the counts in the pocket, we then compute the address in a second storage area at which the elements which contain this bit pattern will be stored. Then we move each element into the place indicated by the address calculation and add one to the address table. In our implementation of this algorithm we provide an introductory program which calculates the optimum size pocket for the number of elements being processed. Radix sort requires twice as much storage as QKSORT and consists of three parts: counting of the number of elements in a pocket, address calculation and moving of

the elements into the new array. This process is repeated as many times as the pocket size fits in the word length. For example, if the pocket size was 16, the procedure would be executed four times.

On the Cray X1, for large numbers of elements, say 10,000, the address calculation uses the most computer time.

```
word[j+1]  = word[j+1] + word[j]
```

This is because the compiler brings each of the words into S registers and stores back from S registers. We can improve this code by the loading L words in a vector register, calculating the sums in the S registers and storing from the S register. Storing back from a V register takes more computer time. We call this the V-S method.

There is also another way to make this calculation. We create two loops as follows where n is the number of elements to be sorted:

```
k = (n – 1)/63 +1

for(j=1; j<k; j++) for(i=0; i<63; i++)
                word[k*i+j+1]  = word[k*i+ j+1] + word[k*i+j]

for(i=1; i<63; i++) for(j=1; j<k; j++)
                word[k*i+j]  = word[k*i+ j] + word[k*i]
```

We call this the 2V method. Note that we use 63 instead of the more natural 64 which is a bad stride for vectors. Table 3 shows how using these two methods improve the code over the ported version. Table 3 uses 53 bit numbers and Table 4 uses 63 bit numbers.

Table 3
Improving PSORT

Count Version		5,000	10,000	50,000	100,000	500,000	1,000,000	4,000,000
Ported		.0049	.0047	.0360	.0722	.4142	.4972	1.2252
V-S Loop		.0013	.0028	.0159	.0327	.1725	.2484	.7838
2V Loops		.0012	.0027	.0092	.0210	.1273	.2015	.7080

We find that PSORT runs faster than QKSORT for approximately 500 elements on a T90 and an X1. Table 4 compares the X1 with the other computers. We note that the X1 is slower than the T90. We believe this to be due to the compiled code and hope that future compiler improvements will solve this problem. The other machines do not have vectors and on these machines we usually use another library version of radix sort.

MSORT

MSORT is a multiprocessor sort which was developed for the T3D. This particular sort starts with the same number of elements in each processor and ends with the same number of elements in each processor. The lowest numbered processor then has the smallest elements.

Table 4
PSORT in 2003

Size Machine		10,000	50,000	100,000	500,000	1,000,000	5,000,000	10,000,000
T3E		.037	.377	.747	.3.307	6.371		
T90		**.004**	**.014**	**.024**	**.109**	**.170**	**.802**	**1.600**
Alpha EV6.7		.004	.081	.211	1.447	2.921	16.749	37.436
Alpha EV6.8		.003	.026	.077	.431	1.043	9.158	19.630
X1		.003	.011	.023	.133	.215	1.024	2.002

There are three phases to this sort. First each processor uses either QKSORT or PSORT to sort the elements in its own buffer. Then information about the number of elements in each 1/8 of the memory is passed to the head processor. After some calculation and requests for more detailed information, ranges are assigned to each processor. In the third phase, each processor sends the proper elements to the other processors. Each processor does a merge on the data passed to it. As the number of processors goes up, this phase becomes more time consuming.

On the X1, SHMEM is used for communication between processors. On the T3E we use QKSORT for the first phase, On the X1 it seems better to use PSORT since it is faster and as double memory size is required for the merge phase no extra memory is needed. Table 5 compares the T3E and X1 for various numbers of processors. The limit of 24 processors is used only because that

was the number of X1 processors available at the time of the tests. The table points out a start up cost and that the SHMEM on the X1 is slower since we know that the transfer speed and the sort times are faster on the X1. These SHMEM times have improved since January when this effect was first noticed but there needs to be more work in this area.

Table 5 - MSORT in 2003

Size	10,000	100,000	1,000,000	10,000,000
T3E - 2	.003	.040	.488	
T3E - 4	**.002**	**.023**	.263	
T3E - 8	.002	.025	.142	1.661
T3E - 16	.004	.068	.099	1.098
T3E - 24	.009	.085	**.074**	**.755**
X1 - 2	**.013**	.091	.955	
X1 - 4	.014	.064	.650	
X1 - 8	.033	.050	.492	4.011
X1 - 16	.042	.048	.348	2.292
X1 - 24	.109	**.026**	**.191**	**1.674**

Conclusions

Though we had hoped to use the improvement of these programs as a starting point to write new multistreaming versions, this has not happened. We are planning a summer program which includes looking at new sorting techniques. However, the question remains as to whether it makes sense to create multistreaming versions of these codes considering that multistreaming may not be available on future architectures. However, we have already considered some interesting ideas for using MSPs rather than one SSP.

Much has been gained from spending time on these sorting programs. First, we helped critique the X1 software and send information to Cray. We created a good set of sorting routines for the X1. We found areas where the compilers needed to be improved – some of those improvements have been made, some are being considered. We learned some of the great features of the X1 such as memory bandwidth, additional registers and the ability to carry out certain operations in different types of registers thereby reducing the movement of variables from one type of register to another. Sorting remains important for research and production.

[1] Robert Sedgewick, Algorithms in C, Addison-Wesley, 1990.