Parallelization of a Radix Sort Algorithm

Calebe de Paula Bianchini
calebe.bianchini@mackenzie.br

1. The Problem Description

Given a set of unsorted items with keys that can be considered as a binary representation of an integer, the bits within the key can be used to sort the set of items. This method of sorting is known as Radix Sort.
Write a program that includes a threaded version of a Radix Sort algorithm that sorts the keys read from an input file, then output the sorted keys to another file. The input and output file names shall be the first and second arguments on the command line of the application execution.
The first line of the input text file is the total number of keys (N) to be sorted; this is followed by N keys, one per line, in the file. A key will be a seven-character string made up of printable characters not including the space character (ASCII 0x20). The number of keys within the file is less than 2^31 - 1. Sorted output must be stored in a text file, one key per line.
Timing: If you put timing code into your application to time the sorting process and report the elapsed time, this time will be used for scoring. If no timing code is added, the entire execution time (including time for input and output) will be used for scoring.

2. The Serial Solution

The Radix Sort algorithms can be classified in two basic groups [1, 2]: least significant digit (LSD) and most significant digit (MSD). The LSD approach examines the digits in the keys in a right-to-left order, working with the least significant digits. The other approach (MSD) examines the digits in the key in a left-to-right order. Figure 1 [3] show an example of LSD and MSD radix-sort using 3-digits integer keys.

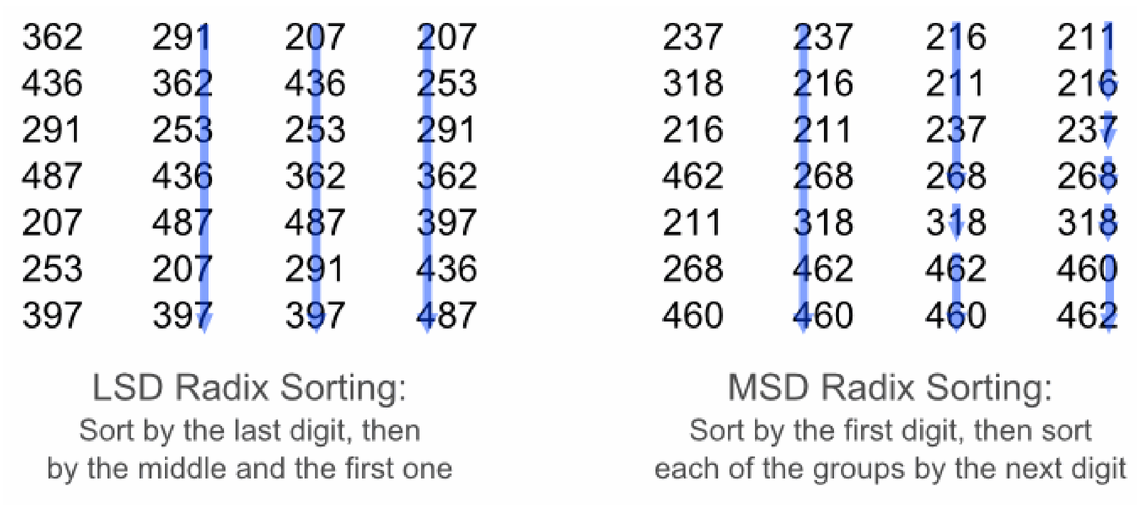

Figure 1. Summary of LSD and MSD approaches.

When a digit is chosen for MSD radix-sort (e.g. 2nd column of MSD picture), an internal sorting is used to organize the keys [1], grouping them with the same digit.

The serial solution proposed uses a MDS radix-sort algorithm to sort the keys and uses a "fat-pivot" quicksort algorithm (three-way partitioning) [1, 4] for internal sorting, taking advantage of repeating digits.

3. The First Parallel Solution Attempt

Studying carefully the MSD approach, it is possible to realize that, when the first significant digit is ordered (1st step), all the keys are partitioned by its first digit; when the second significant digit is ordered (2nd step), the keys are partitioned again.
Considering that each partition is a job and a job can be allocated for one processor (without race condition), it is possible to project a parallel algorithm for radix-sort. For thread-control, the solution uses OpenMP [4], but still uses POSIX mutex and semaphores for IPC and critical section.
The idea is creating a job every time a partition is identified. This job is put into a pool of jobs. Each processor retrieves jobs from the pool (like producer/consumer [5]) and sorts the part-keys. In this part-sorting, more jobs are created because other partitions are identified (see Figure 1). When the pool of jobs is empty, there are no jobs anymore and the entire keys are sorted, stopping the algorithm.
It would be an interesting approach if there were not a critical section (producer/consumer – mutex/semaphore). The critical section is a bottleneck for this solution. Moreover, the sorting algorithm is very fast for a small amount of keys (+/- 1,000,000), being faster than these producer/consumer approach.
Table 1 show a simple time comparison for the serial solution and a dual-process parallel solution (the external-time means total time and the parentheses-time indicates the first most significant digit sort time).

Table 1. Simple time comparison (in seconds).

Number of keys	Serial Solution	1st Parallel Solution (2 proc.)
1,000	0.000487	0.010744 (0.000369)
100,000	0.063586	0.525342 (0.025772)
500,000	0.389194	73.858603 (0.120917)
1,000,000	0.832710	879.922349 (0.218579)
50,000,000	47.611455	ω (11.061750)
100,000,000	104.441001	ω (23.215623)

4. The Final Parallel Solution

The final parallel solution uses the same idea from the last parallel solution. Beyond the normal threads, an additional thread is created only for synchronization: it will remain sleeping until all the other threads reach an end (similar to reader/writer problem [5]). This thread synchronization guarantees the values to the output file.
Analyzing the input (ASCII from 0x21 to 0x7E) [6], there are only 94 printable characters. In other words, only 94 partition after sorting the 1st most significant digit. Considering a balanced distribution, each of these partitions will contain +/- 22,845,570 keys and based on the jobs problem from Table 1, it is not a good idea to have more jobs. So, after the creation of these 94 jobs, each thread sorts the keys on the partition by itself. If these keys are unbalanced, there is a threshold that indicates to create a job is a partition has more than 22,845,570 keys.

Besides, the first job, for the first most significant digit, begins to be the slowest part of the execution for a number of keys above 100,000,000. The iterative quicksort has to be improved and checkpoints are created. These checkpoints indicate that the quicksort already have sorted part of the total keys. After that, the partitions until the checkpoint are verified and all the jobs created. That's mean: from each checkpoint, new jobs are put on the pool for the parallel-thread sorting.
This improvement for quicksort considers the use of virtual memory in the hardware/software specification, too. All the jobs created sorts first the first's elements of the array, avoiding too much swap from the disk to the memory (2^31-1 keys, with 7+1 character length will consume at least 16Gbytes of RAM).

Table 2 show the values collected from an 8-processor machine for a time comparison between the algorithms.

Table 2. Comparison between algorithms (in seconds).

Number of keys	Serial Solution	Parallel Solution		
		2-proc	4-proc	8-proc
1,000	0.000487	0.008024	0.008484	0.105758
100,000	0.063586	0.041309	0.031875	0.104594
500,000	0.389194	0.225825	0.135147	0.128708
1,000,000	0.832710	0.502728	0.289878	0.236469
50,000,000	47.611455	27.055081	15.870333	11.025388
100,000,000	104.441001	59.439492	33.920247	23.915724
500,000,000	511.607666	287.287585	166.408876	122.619186
1,000,000,000	1174.523109	595.044868	355.776813	263.528890

5. Conclusion

Sorting is the most common task in computer. Because ordered data are easy to manipulate, many software require sorted data.
But, the amount of the data sorted is fundamental to construct an efficient parallel sort algorithm [7]. The data have to be organized on memory and the processes have to manipulate it in a coordinate manner.
The results from last section show that a parallel algorithm is easy to project, but it is not so easy to get a linear speedup (see Table 2). In general, considering a good sort algorithms, a small amount of data sorted in a serial execution is faster than the same approach with a parallel algorithm. Thus, a good approach is to consider the amount of data to decide the way to sort: serial or parallel.

6. References

[1] Sedgewick, R. *Algorithms in Java: Parts 1-4*. 3th edition. Addison-Wesley, 2002.
[2] Wikipedia. *Radix Sort*. URL: http://en.wikipedia.org/wiki/Radix_sort . Accessed in 04/2009.
[3] Figure 1. *RadixSort*. URL: http://www.strchr.com/_media/radixsort.png . Accesses in 04/2009.

[4] Wikipedia. *Quicksort*. URL: http://pt.wikipedia.org/wiki/Quicksort . Accessed in 04/2008.
[4] OpenMP. *OpenMP.org*. URL: http://openmp.org/wp/ . Accessed in 04/2009.
[5] Tanembaum. A. *Modern Operation Systems*. Prentice-Hall, 2007.
[6] Wikipedia. ASCII. URL: http://en.wikipedia.org/wiki/ASCII . Accessed in 04/2009.
[7] Grama, A; Gupta, A; Karypis, G; Kumar, V. *Introduction to Parallel Computing*. Addison-Wesley, 2003.