ANALYSIS OF THE PERFORMANCE OF THE PARALLEL QUICKSORT METHOD

D. J. EVANS and NADIA Y. YOUSIF

Department of Computer Studies,
Loughborough University of Technology,
Loughborough, Leicestershire, LE 11 3TU, England

Dept. of Computer Science,
College of Science,
Basrah University, Iraq

Abstract.

In this paper a theoretical analysis of the performance of a parallel form of the Quicksort algorithm is presented and shown to be in qualitative agreement with experiments carried out on the Loughborough University NEPTUNE parallel system.

Keywords: Quicksort, MIMD system, partitioning process, linear insertion.

CR categories: 5.25, 5.31, 6.22.

1. The parallel quicksort algorithm.

The Quicksort algorithm (Hoare [5, 6]) is considered to be the most efficient general-purpose sorting algorithm as both empirical and analytic studies have later proved [7, 8]. Quicksort is a comparative, interchange sorting algorithm and also an example of a distributive sort since it is based on partitioning the original set into smaller subsets.

Sedgewick [9] has proposed the most efficient improvements to Quicksort and also presented the history and a complete survey of the many variants of the Quicksort that have been proposed. He also presented and analysed the proposed algorithm [10,11] which introduced linear insertion to sort the subsets of smaller size and found that the choice of the median of three elements as a partitioning element is better than randomly choosing that element. Thus, these improvements raised the efficiency and slightly reduced the running-time of the algorithm.

Our implementation of the parallel Quicksort is an extension to the algorithms [1] and [3]. Our algorithm is based on finding the median of three elements as a partitioning element. The three elements are the first, middle, and last elements of the set. When the partitioning element, say $V$, is found the original set is partitioned into two subsets $S_1$ and $S_2$ such that all the element of
$S_1$ are less than $V$ and all elements of $S_2$ are greater than $V$. The two subsets $S_1$ and $S_2$ are then sorted by the same procedure if they are sufficiently large. Otherwise, a linear insertion sort procedure is used to sort the subsets. The size of the subset to be sorted by the linear insertion sort should be less than $M$, where $M$ is a constant cf. [9]. The experiments showed that the best value of $M$ is between 5 and 20.

For the partitioning procedure, which is a variant of [9], two queue data structures are used to hold the pointers of the start and the end of the resultant subset in an First-In-First-Out (FIFO) concept. Therefore in order to sort the subset, corresponding start and end pointers are popped from the queue. If this subset is larger than $M$ then it is partitioned and the pointers of the resulting subsets are pushed onto the queue in a similar fashion as before. Otherwise, the subset is sorted by the linear insertion sort procedure and the results are placed in the resultant output vector which is the same as the input vector but its elements are sorted. The input vector of the elements to be sorted and two queues are kept in the shared memory in order to be accessed by all the processors.

In Sedgewick's algorithm small subfiles are ignored during the partitioning stages and a single insertion sort is applied at the end. In the above variant, the small subfiles are treated as soon as they pop up from the queues introducing the advantageous fact that some insertion sorting can be carried out in parallel with the partitioning.

In fact, the nature of this algorithm makes it highly suitable to be implemented on a parallel MIMD computer. Since the partitioning procedure is continuously proceeding to produce numbers of subsets of smaller sizes of the current considered set, therefore, these subsets can be sorted simultaneously as long as free processors are available.

In the parallel implementation only one path is generated first to partition the original set into two subsets which are placed on these two subsets, i.e. two paths are generated to yield four subsets. The partitioning of the two subsets is performed simultaneously by two processors. Next, the four subsets are partitioned by four processors to yield eight subsets and so on until there are no further subsets to partition. If we assume that the subsets produced by the partitioning procedure are of equal size then, if the size of the original set is $N$, the size of the subsets in the subsequent steps of the algorithms will be $N/2$, $N/4$ and so on.

This algorithm illustrates that the number of parallel paths generated in each step depends on the number of subsets produced by the previous step which are kept in the queue. After some steps, the number of the generated paths will be greater than the number of the available processors. In this case, the first $P$ subsets, where $P$ is the number of available processors, are processed and the others will be held in a queue to wait for a free processor. The step is completed.