æ

Parallel merge sort on shared memory multicomputer

V. Havran, P. Šimek, M. Šoch, P. Tvrdík

CTU, Fac. of Electrical Eng., Dept. of Computer Science and Engineering
Karlovo nám. 13, 121 35 Praha 2

Key words: sorting, parallel algorithm, shared memory

1 Introduction

This paper deals with Cole's Merge Sort algorithm based on sequential merge sort algorithm. Due to the requirement to achieve work optimality, the parallel version is a bit more complicated than the *Divide&Conquer* approach of the well known Batcher's Merge Sort. The main goal of our work was to implement the algorithm on a shared memory machine, test it, and evaluate the performance.

2 Sequential Merge Sort

This algorithm is very well-known. The solution is based on a recursive approach. The input data sequence is divided to two subsequences. Subsequences are recursively sorted and then merged together into one sequence. Recursion ends when the subsequences have two elements only - they are sorted trivially by compare and exchange operation.

Sequential complexity of merging two subsequences of n elements is $\Theta(n)$. The complexity of the whole sequential Merge Sort for n data sequence is $\Theta(n \log n)$.

3 Cole's Merge Sort

This section shows the idea behind the basic recursive step of Cole's Merge Sort. The input of the algorithm is two sorted sequences `L` and `R` and auxiliary sequence `EM` made by merging elements from even positions of sequences `L` and `R`. The output is sorted sequence `M`.

The idea of the algorithm is based on the fact that is possible to precalculate deterministicaly positions in output sequence `M` of elements at odd positions in input sequences `L` and `R`. The output positions can be computed independently without conflicts by a set of processors.

The algorithm is scalable, its time complexity is $O(\log^2 n)$ on $\Omega(\frac{n}{\log n})$ processors and it is cost optimal for $\Theta(\frac{n}{\log n})$ processors.

4 Implementation

During implementation of the algorithm we encountered and had to fix several interesting issues. Merge Sort assumes input sequence of length $n = 2^k$ for some integer $k > 0$. On real parallel machines, the number of processors, p, is always $p \ll n$ and p is not necessarily a power of 2. This makes no problem in the ascending phase where we precompute the final positions of numbers at odd positions in `L`, `R` and merge 2 sorted sequences, since the standard Cole's algorithm is fully scalable (all computations on individual elements of arrays are independent). However, in the descending phase, we end up with $2^{\lceil \log p \rceil}$ unsorted *blocks* of $2^{k - \lceil \log p \rceil}$ elements that should be sorted by p processors. A trivial solution is to distribute blocks evenly among processors so that each is assigned 1 or 2 blocks and sorts them sequentially. Unfortunately, some processors have twice as much of work as the other ones. Here, we propose an optimized solution for this sequential step, in which all processors are always loaded uniformly.

The aim of the sequential part is to transform the input unsorted sequence of 2^k elements into $2^{\lceil \log p \rceil}$ sorted blocks of size $2^{k-\lceil \log p \rceil} = 2^{k'}$. The number of compare-exchange operations is $2^{\lceil \log p \rceil} \cdot \sum_{i=1}^{k'-1} (2^i - 1) \cdot 2^{k-i} = 2^{\lceil \log p \rceil} \cdot ((k'-2) \cdot 2^{k'} + 2) = 2^{\lceil \log p \rceil} \cdot ((k - \lceil \log p \rceil - 2) \cdot 2^{k-\lceil \log p \rceil} + 2) = 2^k \cdot (k - \lceil \log p \rceil - 2) + 2^{\lceil \log p \rceil + 1}$.

If p is a power of 2, then $2^{\lceil \log p \rceil} = p$, each processor is assigned exactly one block and sorts it sequentially.

Assume now that p is not a power of 2. We proceed as follows: The input sequence of 2^k elements is partitioned evenly into p subsequences s_j, $1 \leq j \leq p$, of sizes $2\lceil 2^k/(2p) \rceil$ or $2\lfloor 2^k/(2p) \rfloor$, processor P_j is assigned s_j. Sorting is done in $k - \lceil \log p \rceil$ steps. Before step i, $1 \leq i \leq k - \lceil \log p \rceil - 1$, each s_j consists of sorted blocks of size 2^{i-1}, where the leftmost (rightmost) blocks may be cut by the boundaries between s_j and s_{j-1} (s_{j+1}, respectively). In step i, pairs of blocks of size 2^{i-1} are merged into sorted blocks of size 2^i. If given block of size 2^i belongs to only one subsequence s_j, then merging is done sequentially by processor P_j. If a block of size 2^i is cut by the boundary between 2 adjacent sequences, say s_j and s_{j+1}, $1 \leq j \leq p-1$, then the merge is done cooperatively of processors P_j and P_{j+1} as depicted on Fig. 1. Processor P_j fills its left part of new blocks by taking elements rightward, whereas P_{j+1} fills the right part leftward. The concurrent merge is conflict-free. As a result, all processors are uniformly loaded in every step of the sequential part.

The algorithm was implemented in C language using SHM library for shared memory management.

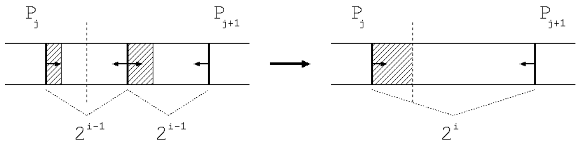

Fig. 1: Merging by two processors P_j and P_{j+1}.

5 Conclusion

We have modified Cole's Merge Sort by optimizing its sequential part for arbitrary number of processors. The algorithm was implemented and we have done some preliminary measurements. The measurements have bee done on the Digital Alpha Server 8400 in ZCU Pilsen on up to 8 processors. On average, the speedup was about 45% for the standard Cole's algorithm and over 50% for our optimized modification.

References:

[1] Cole, R.: *Parallel Merge Sort.* SIAM Journal of Computing, 17(4): 770-785, 1988.

[2] JáJá, J.: *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[3] Havran, V.: *The Simulation of Optical Effects.* Master Thesis, DCSE CTU Prague, 1996.