CCIT Report #802 January 2012

Merge Path – Cache-Efficient Parallel Merge and Sort

Saher Odeh, Oded Green ‡, Zahi Mwassi, Oz Shmueli, Yitzhak Birk
Electrical Engineering Department
Technion
Haifa, Israel
{sahero, ogreen, zahim}@tx.technion.ac.il, {shmueli, birk}@ee.technion.ac.il

***Abstract*—Merging two sorted arrays is a prominent building block for sorting and other functions. Its efficient parallelization requires balancing the load among compute cores, minimizing the extra work brought about by parallelization, and minimizing inter-thread synchronization requirements. Due to the extremely low compute to memory-access ratio, it is also critically important to efficiently utilize the memory system: minimize memory traffic, maximize the cache hit rate and minimize cache-coherence related activity.**

We present a novel approach to partitioning the two sorted arrays into pairs of contiguous sequences of elements, one from each array, such that 1) each pair comprises any desired total number of elements, and 2) the elements of each pair form a contiguous sequence in the final merged sorted array. While the resulting partition and the computational complexity are similar to those of certain previous algorithms, our approach is different, extremely intuitive, and offers interesting insights. Based on this, we present a synchronization-free, cache-efficient merging (and sorting) algorithm. While we use CREW PRAM as the basis, our algorithm is easily adaptable to additional architectures. In fact, our approach is even relevant to sequential cache-efficient sorting. The new algorithm has been implemented both on the HyperCore many-core shared-cache architecture and on a sizable x86 system, with emphasis on cache efficiency. The algorithms and performance results are presented, along with important cache-related insights.

Keywords-component; Cache Memories; Parallelism and concurrency; Parallel processors; Sorting and searching

I. INTRODUCTION

Merging two sorted arrays, A and B, to form a sorted array S is an important utility, and is the core the of merge-sort algorithm [1]. The merging (e.g., in ascending order) is carried out by repeatedly comparing the smallest (lowest-index) as-yet unused elements of the two arrays, and appending the smaller of those to the result array.

Given an (unsorted) N-element array, merge-sort comprises a sequence of $\log_2 N$ rounds: in the first round, $N/2$ disjoint pairs of adjacent elements are sorted, forming $N/2$ sorted arrays of size two. In the next round, each of the $N/4$ disjoint pairs of two-element arrays is merged to form a sorted 4-element array. In each subsequent round, array pairs are similarly merged, eventually yielding a single sorted array.

Consider the parallelization of merge-sort using p compute cores (or processors or threads, terms that will be used synonymously). Whenever $|S| = N \gg p$, the early rounds are trivially parallelizable, with each core assigned a subset of the array pairs. This, however, is no longer the case in later rounds, as only few arrays remain. Because the total amount of computation is the same for all rounds, effective parallelization thus requires the ability to parallelize the merging of two sorted arrays.

An efficient Parallel Merge algorithm must have several salient features, some of which are required due to the very low compute to memory-access ratio: 1) equal amounts of work for all cores; 2) minimal inter-core communication (platform-dependent ramifications); 3) minimum excess work (for parallelizing, as well as replication of effort); and 4) efficient access to memory (high cache hit rate and minimal cache-coherence overhead). Coherence issues may arise due to concurrent access to the same address, but also due to concurrent access to different addresses in the same cache line (false sharing). Memory issues have platform-dependent manifestations.

The naïve approach to parallel merge entails partitioning each of the two arrays into equal-length contiguous sub-arrays and assigning a pair of same-numbered sub arrays to each core. Each core then merges its pair to form a single sorted array, and those are concatenated to yield the final result. Unfortunately, this is incorrect. (To see this, consider the case wherein all the elements of A are greater than all those of B.) So, correct partitioning is the key to success.

In this paper, we present a parallel merge algorithm for Parallel Random Access Machines (PRAM), namely shared-memory architectures that permit concurrent (parallel) access to memory. PRAM systems are further categorized as CRCW, CREW, ERCW or EREW, where C, E, R and W denote concurrent, exclusive, read and write, respectively. Our algorithm assumes CREW, but can be adapted. Also, complexity calculations assume equal access time of any core to any address, but this is not a requirement.

Our algorithm is load-balanced, lock-free, requires negligible excess work, and is extended to a memory-efficient version. Being lock-free, the algorithm does not rely on a set of atomic instructions of any particular platform. The efficiency of memory access is also not confined to one kind of architecture; in fact, the memory access is efficient for both private- and shared-cache architectures.

‡*Oded Green is currently with the School of Computational Science and Engineering at Georgia Tech, GA 30332. This work was done while Oded was at the Technion.*