2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

Merge Path - Parallel Merging Made Simple

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. Efficient use of memory is also important.**

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 algorithms and performance results are presented, along with important cache-related insights.

Keywords-component; 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 Coherence issues may arise due to concurrent access to the same address. Memory issues have platform-dependent manifestations.

A naïve approach to parallel merge would entail 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 a negligible amount of 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.*

978-0-7695-4676-6/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPSW.2012.202

We show a correspondence between the merge operation and the traversal of a path on a grid, from the upper left corner to the bottom right corner and going only rightward or downward. This greatly facilitates the comprehension of parallel merge algorithms. By using this path, dubbed *Merge Path*, one can divide the work equally among the cores.

Our actual basic algorithm is similar to that of [2], but is more intuitive and conceptually simpler.

The remainder of the paper is organized as follows. In Section II, we present the Merge Path, the Merge Matrix and the relationship between them. These are used in Section III to develop parallel merge and sort algorithms. Section IV introduces cache-related issues and presents a cache-efficient merge algorithm. Section V discusses related work, Section VI presents implementations and performance results, and Section VII offers concluding remarks.

II. MERGE PATH

A. Construction and basic properties

Consider two sorted arrays, A and B, with $|A|$ and $|B|$ elements, respectively. Without loss of generality, assume that they are sorted in ascending order. As depicted in Fig. 1 (ignore the contents of the matrix), create a column comprising A's elements and a Row comprising B's elements, and an $|A|x|B|$ matrix M, each of whose rows (columns) corresponds to an element of A (B). We refer to this matrix as the *Merge Matrix*. A formal definition and additional details pertaining to M will be provided later.

Next, let us merge the two arrays: in each step, pick the smallest (regardless of array) yet-unused array element. Alongside this process, we construct the *Merge Path*. Referring again to Fig. 1, start in the upper left corner of the grid, i.e., at the upper left corner of *M[1,1]*. If *A[1]>B[1]*, move one position to the right*;* else move one position downward. Continue similarly as follows: consider matrix position *(i,j)* whose upper left corner is the current end of the merge path: if *A[i]>B[j]*, move one position to the right*;* else move one position downward; having reached the right or bottom edge of the grid, proceed in the only possible direction. Repeat until reaching the bottom right corner.

The following four lemmas follow directly from the construction of the Merge Path:

Lemma 1: Traversing a Merge Path from beginning to end, picking in each rightward step the smallest yet-unused element of *B,* and in each downward step the smallest yet-unused element of *A*, yields the desired merger. □

Lemma 2: Any contiguous segment of a Merge Path is composed of a contiguous sequence of elements of *A* and of a contiguous sequence of elements of *B*. □

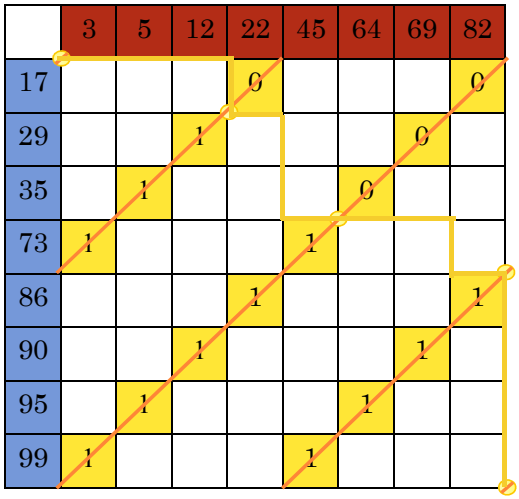

Figure 1 - The cross diogonals in a Merge Matrix are used to find the points of change between the ones and the zeros, i.e., the intersections with the Merge Path.

Lemma 3: Non-overlapping segments of a merge path are composed of disjoint sets of elements, and vice versa.□

Lemma 4: Given two non-overlapping segments of a merge path, all array elements composing the later segment are greater than or equal to all those in the earlier segment. □

Theorem 5: Consider a set of element-wise disjoint sub-array pairs (one, possibly empty sub-array of *A* and one, possibly empty sub-array of *B*), such that each such pair comprises all elements that, once sorted, jointly form a contiguous segment of a merge path. It is claimed that these array pairs may be merged in parallel and the resulting merged sub-arrays may be concatenated according to their order in the merge path to form a single sorted array.

Proof: By Lemma 1, the merger of each sub-array pair forms a sorted sub-array comprising all the elements in the pair. From Lemma 2 it follows that each such sub-array is composed of elements that form a contiguous sub-array of their respective original arrays, and by Lemma 3 the given array pairs correspond to non-overlapping segments of a merge path. Finally, by Lemma 4 and the construction order, all elements of a higher-indexed array pair are greater than or equal to any element of a lower-indexed one, so concatenating the merger results yields a sorted array. □

Corollary 6: Any partitioning of a given Merge Path of input arrays *A* and *B* into non-overlapping segments that jointly comprise the entire path, followed by the independent merger of each corresponding sub-array pair and the concatenation of the results in the order of the corresponding Merge-Path segment produces a single sorted array comprising all the elements of *A* and *B*. □

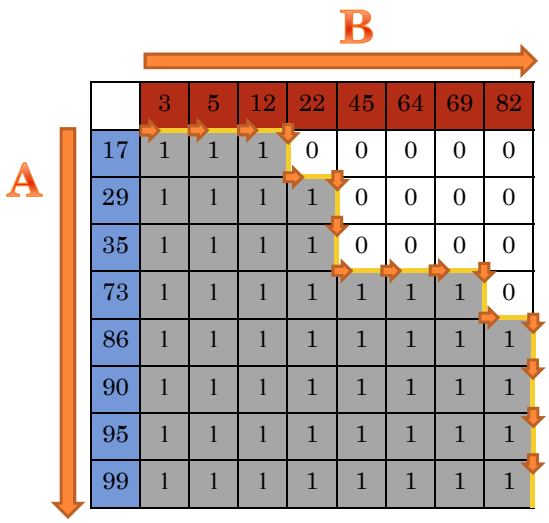

Figure 2 - Merge Matrix and Merge Path.

Corollary 7: Partitioning a Merge Path into equisized segments and merging the corresponding array pairs in parallel balances the load among the merging processors.

Proof: each step of a Merge Path requires the same operations (read, compare and write), regardless of the outcome. □

Equipped with the above insights, we next set out to find an efficient method for partitioning the Merge Path into equal segments. The challenge, of course, is to do so without actually constructing the Merge Path, as its construction is equivalent to carrying out the entire merger. Once again using the geometric insights provided by Fig. 1, we begin by exposing an interesting relationship between positions on any Merge Path and cross diagonals (ones slanted upward and to the right) of the Merge Matrix *M*. Next, we define the contents of a Merge Matrix and expose an interesting property involving those. With these two building blocks at hand, we construct a simple method for parallel partitioning of any given Merge Path into equisized segments. This, in turn, enables parallel merger.

B. The Merge Path and cross diagonals

Lemma 8: Regardless of the route followed by a Merge Path, and thus regardless of the contents of *A* and *B*, the *i*'th point along a Merge Path lies on the *i*'th cross diagonal of the grid and thus of the Merge Matrix *M*.

Proof: each step along the Merge Path is either to the right or downward. In either case, this results in moving to the next cross diagonal. □

Theorem 9: Partitioning a given merge path into *p* equisized contiguous segments is equivalent to finding its intersection points with *p-1* equispaced cross diagonals of *M*,

Proof: follows directly from Lemma 8. □

C. The Merge Matrix – content & properties

Definition 1: A binary merge matrix M of A, B is a Boolean two dimensional matrix of size $|A| \times |B|$ such that:

$$M[i,j] = \begin{cases} 1 & A[i] > B[j] \\ 0 & otherwise \end{cases}.$$

Proposition 10: Let M be a binary merge matrix. Then $[i,j] = 1 \Rightarrow \forall k,m: i \le k \le |A| \wedge 1 \le m \le j, M[k,m] = 1$

Proof: If $M[i,j] = 1$ then according to definition 1, $A[i] > B[j]$. $k \ge i \Rightarrow A[k] \ge A[i]$ (A is sorted). $j \ge m \Rightarrow B[j] \ge B[m]$ (B is sorted). $A[k] \ge A[i] > B[j] \ge B[m]$ and according to definition 1, $M[k,m] = 1$. □

Proposition 11: Let M be a binary merge matrix. If $M[i,j] = 0$, then $\forall k,m: 1 \le k < i \wedge j \le m \le |B|$, $M[k,m]$ is false.

Proof: Similar to the proof of proposition 10. □

Corollary 12. The entries along any cross diagonal of *M* form a monotonically non-increasing sequence. □

D. The Merge Path and the Merge Matrix

Having established interesting properties of both the Merge Path and the Merge Matrix, we now relate the two, and use *P(M)* to denote the Merge Path corresponding to Merge Matrix *M*.

Proposition 13: Let (i,j) be the highest point on a given cross diagonal M such that $M[i,j-1] = 1$ if exists, otherwise let (i,j) be the lowest point on that cross diagonal. Then, $P(M)$ passes through (i,j). This is depicted in Figure 2.

Proof: by induction on the points on the path.

Base: The path starts at $(1,1)$. The cross diagonal that passes through $(1,1)$ consists only of this point; therefore, it is also the lowest point on the cross diagonal.

Induction step: assume the correctness of the claim for all the points on the path up to the point (i,j). Consider the next point on $P(M)$. Since the only permissible moves are R, D, the next point can be either $(i,j+1)$ or $(i+1,j)$, respectively.

Case 1: R move. The next point is $(i,j+1)$. According to Definition 1, $M[i,j] = 1$. According to the induction assumption, either $i = 1$ or $M[i-1,j] = 0$. If $i = 1$ then the new point is the highest point on the new cross diagonal such that $M[i,j] = 1$. Otherwise, $M[i-1,j] = 0$. According to Proposition 11, $M[i-1,j+1] = 0$. Therefore, $(i,j+1)$ is the highest point on its cross diagonal at which $M[i,j] = 1$.

Case 2: the move was D, then the next point is $(i+1,j)$. According to Definition 1, $M[i,j] = 0$. According to the induction assumption, either $j = 1$ or $M[i,j-1] = 1$. If $j = 1$ then the new point is the lowest point in the new cross diagonal. Since $M[i,j] = 0$ and according to Proposition 11, the entire cross diagonal is 0. Otherwise,

$M[i, j-1] = 1$. According to Proposition 10, $M[i+1, j-1] = 1$. Therefore, $(i, j+1)$ is the highest point on its cross diagonal at which $M[i+1, j-1] = 1$. □

Theorem 14: Given sorted input arrays *A* and *B*, they can be partitioned into *p* pairs of sub-arrays corresponding to *p* equisized segments of the corresponding merge path. The *p-1* required partition points can be computed independently of one another (optionally in parallel), in at most $\log_2(\min(|A|,|B|))$ steps per partition point, with neither the matrix nor the path having actually been constructed.

Proof: According to Theorem 9, the required partition points are the intersection points of the Merge Path with *p-1* equispaced (and thus content-independent) cross diagonals of *M*. According to Corollary 12 and Proposition 13, each such intersection point is the (only) transition point between '1's and '0's along the corresponding cross diagonal. (If the cross diagonal has only '0's or only '1's, this is the uppermost and the lower most point on it, respectively.) Finding this partition point can be done by way of a binary search, whereby in each step a single element of *A* is compared with a single element of *B*. Since the length of a cross diagonal is at most $\min(|A|,|B|)$, at most $\log_2(\min(|A|,|B|))$ steps are required. Finally it is obvious from the above description that neither the Merge Path nor the Merge Matrix needs to be constructed and that the *p-1* intersection point can be computed independently and thus in parallel. □

III. Parallel Merge and Sort

Given two input arrays *A* and *B* parallel merger is carried out by *p* processors as follows:

Algorithm 1 – Parallel Merge (*A,B,p*)

/* *i* = processor number in the range 1..p */

In parallel do:

1. DiagonalNum=*(i-1)*·(|*A*|+|*B*|)/*p*+1
2. Compute intersection of the merge path with the relevant diagonal //Binary search
3. Execute (|*A*|+|*B*|)/*p* steps of sequential merge, writing the results to output array positions starting at *(i-1)*· (|*A*|+|*B*|)/*p*+1

Barrier;

Remark. Note that no communication is required among the cores: they write to disjoint sets of addresses and, with the exception of reading in the process of finding the intersections between the Merge Path and the diagonals, read from disjoint addresses. Whenever |*A*|+|*B*|>>*p*, which is the common case, this means that concurrent reads from the same address are rare.

Summarizing the above, the time complexity of the algorithm for |*A*|+|*B*|=*N* and *p* processors is given by $O(N/p + \log(N))$, and the work complexity is given by $O(N + p \cdot \log N)$. For $p < N/log(N)$, this algorithm is considered to be optimal. In the next section, we will further address the issue of efficient memory (cache) utilization.

Finally, merge-sort can be used, employing Parallel Merge to carry out each of $\log_2 N$ rounds. The rounds are carried out one after the other.

The time complexity of this Parallel Merge-Sort is:

$$O(N/p \cdot log(N/p) + N/p \cdot log(p) + \log(p) \cdot \log(N)) = O(N/p \cdot \log(N) + \log(p) \cdot \log(N))$$

In the first expression, the first component corresponds to the sequential sort carried out concurrently by each core on *N/p* input elements, and the two remaining ones correspond to the subsequent rounds of parallel merges.

IV. Cache Efficient Merge Path

A. Overview

The rate at which merging and sorting can be performed even in memory (as opposed to disk), is often dictated by the performance of the memory system rather than by processing power. This is due to the fact that these operations require a very small amount of computing per unit of data, and the fact that only a small amount of memory, the cache, is reasonably fast. (The next level in the memory hierarchy typically features a ten-fold higher access latency as well as coarser memory-management granularity.) Parallel implementation on a shared memory system further aggravates the situation for two reasons: 1) the increased compute power is seldom matched by a commensurate increase in memory bandwidth, at least beyond the 1st-level or 2nd-level cache, and 2) cache coherence mechanisms can present an extremely high overhead. In this section, we address the memory issues.

Assuming large arrays (relative to cache size) and merge-sort, it is clear that data will have to be brought in multiple times ($\log_2 N$ times, one for each level of the merge tree, for non cache oblivious algorithms), so we again focus on merging a pair of sorted arrays.

In the remainder of this section, we examine the cache efficiency issue in conjunction with our algorithm, offering important insights, exploring trade-offs and presenting our approaches.

B. Cache-Efficient Parallel Merge

In this sub-section we present an extension to our algorithm for parallel merging that is also cache-efficient. It is stated in the context of a PRAM-like system with a shared-memory hierarchy (including a shared cache).

Collisions in the cache between any two items are avoided when they are guaranteed to be able to reside in different cache locations, as well as when they are guaranteed to be in the cache at different times. In a Merge

operation, a cache-resident item is usually required for a very short time, and is used only once. However, many items are brought into the cache. Also, the relative addresses of "active" items are data dependent. This is true among elements of different arrays (*A, B, S)* and, surprisingly, also among same-array elements accessed by different cores. This is because the segment-partition points in any given array are data dependent, as is the rate at which its elements are consumed.

Given our efficient parallelization, we are able to efficiently carry out parallel merger of even cache-size arrays. In view of this, we explore approaches that ensure that all elements that may be active at any given time can co-reside in cache.

Let *C* denote cache size (in elements). Our general approach is to break the overall merge path into cache-size (actually a fraction of that) segments, merging those segments one after the other, with the merging within each segment being parallelized. We refer to this as *Segmented Parallel Merge*, SPM. See Fig. 3.

Lemma 15. A merge-path segment of length *L* comprises at most *L* consecutive elements of *A* and at most *L* consecutive elements of *B*. □

Theorem 16. Given *L* consecutive elements of *A* and *L* consecutive elements of *B,* starting with the first element of each of them in the segment being constructed, one can compute in parallel the *p* segment starting points so as to enable *p* consecutive segments of length *L/p* to be constructed in parallel.

Proof: Consider the *p-1* cross diagonals of the merge matrix comprising the aforementioned elements of the two arrays, such that the first one is *L/p* away from the upper left corner and the others are spaced with the same stride. The farthest cross diagonal will require the *L*'th provided element from each of the two arrays, and no other point along any of the diagonals will require "later" elements. Also, since the farthest diagonal is at distance *L* from the upper left corner (Manhattan distance), the constructed segment will be of length *L*. □

Remark. Unlike the case of a full merger of two sorted arrays of size *L*, not all elements will be used. While *L* elements will be consumed in the construction of the segment, the mix of elements from *A* and from *B* is data dependent.

In order to avoid the extra complexity of using the same space for input elements and for merged data, let *L=C/3*, where *C* is the cache size.

Algorithm 2 – *Segmented Parallel Merge*

Repeat the following (|*A*|+|*B*|)/*L* times /* *L*=|*C*|/3 */

1. If first iteration, fetch the first *L* items of *A* and *B;* Else fetch the next elements of *A* and *B* in numbers equal to the respective numbers of consumed elements in the previous iteration, overwriting the used elements of the respective arrays (cyclic buffer).

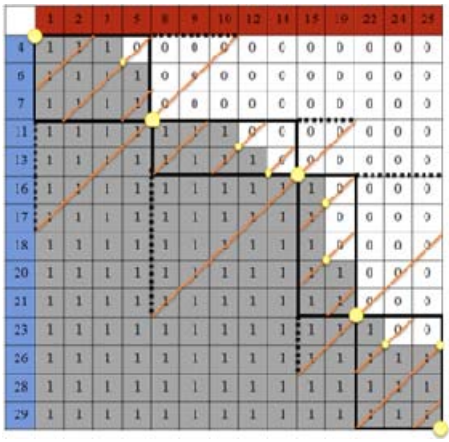

Figure 3 - Merge Matrix for the cache efficient algorithm. The yellow circles depict the initial and final points of the path for a specific block in the cache algorithm.

2. Parallel do:
   a. Find the core's segment starting point /* binary search on cross diagonal */
   b. Merge (sequential) *L/p* steps, commencing at the start point.
3. Write the results out to memory.

Remark. Sufficient total cache size does not guarantee collision freedom (conflict misses can occur). However, we have shown that 3-way associativity suffices to guarantee collision freedom. This will be reported elsewhere.

Computational complexity Assuming a total merged-array segment size of *L=C/3* per sequential iteration of the algorithm, there are *3N/C* such iterations. In each of those, only *2L=2C/3* elements of the input arrays (*L* of each) need to be considered in order to determine the end of the segment and, accordingly, the numbers of elements that should be copied into the cache. Because the sub-segments of this segment are to be created in parallel, each of the *p* cores must compute its starting points (in *A* and in *B*) independently. (We must consider *2L* elements because the end point of the segment, determined by the numbers of elements contributed to it by A and B, is unknown.)

The computational complexity of the cache-efficient merge of *N* elements given a cache of size *C* and *p* cores is:

$$O(N/C \cdot p \cdot logC + N).$$

Normally, *p*<<*C*<<*N*, in which case this becomes *O(N)*. In other words, the parallelization overhead is negligible.

The time complexity is

$$O\big(N/C \cdot (logC + C/p)\big).$$

Neglecting *logC* (the parallelization overhead) relative to *C/p* (the merge itself), this becomes *O(N/p),* which is optimal. Finally, looking at typical numbers and at the

Figure 4 - Cache-efficient parallel sort first stage. Each cache sized block is sorted followed by parallel merging

actual algorithms, it is evident that the various constant coefficients are very small, so this is truly an extremely efficient parallel algorithm and the overhead of partitioning into smaller segments is insignificant.

C. Cache-Efficient Parallel Sort

Initially, partition the unsorted input array into equisized sub-arrays whose size is some fraction of the cache size *C*.

Next, iterate over these sub-arrays, sorting them one by one using the parallel sort algorithm on all p processors as explained in an earlier section.

Finally, proceed with merge rounds; in each of those, the cache-efficient parallel merge algorithm is applied to every pair of sorted sub-arrays. This is repeated until a single array is produced.

We now derive the time complexity of the cache efficient parallel sort algorithm. We divide the complexity into two stages: 1) the complexity of the parallel sorting of the sub-arrays of at most C elements, and 2) the complexity of the cache-efficient merge stages.

In the first stage, depicted in Fig. 4, the parallel sort algorithm is invoked on the cache sized sub-arrays. The number of those sub-arrays is $O(N/C)$. Hence, the time complexity of this stage is $O\big(N/C \cdot (C/p \cdot \log(C) + \log p \cdot \log(C)\big)$.

The second stage may be viewed as a binary tree of merge operations. The tree leaves are the sorted cache sized sub-arrays. Each two merged sub-arrays are connected to the merged sub-array, and so on. The complexity of each level in the tree is $O(N/p + N/C \cdot \log(p))$. The height of the tree is $O(\log N/C)$. Hence, this stage's complexity is $O\big(\log(N/C) \cdot (N/p + N/C \cdot \log(p)\big)$.

The total complexity of the cache-efficient parallel sort algorithm is the summation of the complexities of the two stages, which yield: $O(N/p \cdot \log(N) + N/C \cdot \log(p) \cdot \log(C))$.

One may observe again that the new algorithm has a slightly higher complexity, $N/C \cdot \log(C) \cdot \log(p) > \log \mathrm{N} \cdot \log(p)$, due to the numerous partitioning stages, however for system that a cache miss is expensive, this increase in complexity may be justified.

V. Related Work

In this section, we review previous works on the subjects of parallel sorting and parallel merging, and relate our work to them.

Prior works fall into two categories: 1) algorithms that use a problem-size dependent number of processors, and 2) algorithms that use a fixed number of processors.

Several algorithms have been suggested for parallel sorting. While parallel merge can be a building block for parallel sorting, some of the parallel sorting algorithms do not require merging. An example is Bitonic Sort [4] in which $O(N \cdot (\log N)^2)$ comparators are used ($N/2$ comparators are used in each stage) to sort N elements in $O((\log N)^2)$ cycles. Bitonic sort falls into the aforementioned first category. Our work is in the latter.

We consider two complexity measures: 1) time complexity (the time required to complete the task), and 2) overall work complexity, i.e, the total number of basic operations carried out. In a load balanced algorithm like ours, the work complexity is the product of time complexity and the number of cores. Even with perfect load balancing, however, one must be careful not to increase the total amount of work (overhead, redundancy, etc.), as this would increase the latency. Similarly, one must be careful not to introduce stalls (e.g., for inter-processor synchronization), as these would also increase the elapsed time even if the "net" work complexity is not increased.

Merging two sorted arrays requires $\Omega(N)$ operations. Some of the parallel merging algorithms, including ours, have a work complexity of $O(N + p \cdot \log N)$. For $p \leq N/\log N$, the latter component is negligible and the complexity is $O(N)$, as observed in [5]. Also, there are no synchronization stalls in our algorithm.

In [6], as in our work, a $CREW\ PRAM$ memory model is used. There, a mechanism for partitioning the workload is presented. This mechanism is less efficient than ours and does not feature perfect load balancing; although each processor is responsible for merging $O(N/p)$ elements on average, a processor may be assigned as many as $2N/p$ elements. This can introduce a stall to some of the cores since all the cores have to wait for the heaviest job. For truly efficient algorithms, namely ones in which the constants are also tight, as is the case with our algorithm, such a load imbalance can cause a 2X increase in latency! The time complexity of this algorithm is $O(1 + \log p + \log N + N/p)$. For $N \gg p$, which is the case of interest, it is $O(N/p + \log N)$.

In [5], Akl and Santoro present a merging algorithm that is memory-conflict free using the $EREW$ model. It begins by finding one element in each of the given sorted arrays such that one of those two elements is the median (mid-point) in the output array. The elements found $(A[i], B[j])$ are such that if $A[i]$ is the aforementioned median then $B[j]$ is the largest element of B that is smaller than $A[i]$ or the smallest element of B that is greater than $A[i]$. Once this median point has been found, it is possible to repeat this on both sets of the sub-arrays. Their way of finding the median is similar to the process that we use. The

complexity of finding the median is $O(\log(N))$. As these arrays are non-overlapping, there will not be any more conflict on accessed data. This stage is repeated until there are p partitions. This requires $O(\log(p))$ iterations. Once all the partitions have been found, it is possible to merge each pair of sub-arrays sequentially, concurrently for all pairs, and to simply concatenate the results to form the merged array. The overall complexity of this algorithm is $O(N/p + \log N \log p)$. The somewhat higher complexity is the price for the total elimination of memory conflicts.

In [2], an algorithm that is conceptually similar to that of [5] is presented. They initially present an algorithm that finds one element in each of two given sorted arrays such that one of these elements is $k-th$ smallest element in the output (merged) array. In [5] they start off by finding $k = N/2$. In [2], the elements sought after are those that are equispaced (N/p positions apart) in the output array. Finding each of these elements has the complexity of $O(\log(N))$. This algorithm is aimed for $CREW$ systems. The complexity of this algorithm is $O(N/p + \log N)$.

Our algorithm is very similar to the one presented in [2]. However, our approach is different in that we show a correspondence between finding the desired elements and finding special points on a grid. Finally, using this correspondence along with additional insights and ideas, we also provide cache efficient algorithms for parallel merging and sorting that did not appear in any of the related works.

The work done in [7] is an extension of [2], in which the algorithm is adapted to an $EREW$ machine with a slightly larger complexity of $O(N/P + \log N + \log P)$.

Merging and sorting using GPUs is a topic of great interest as well, and raises additional challenges that need to be addressed. In [8] a radix sort for the GPU is presented. In addition to the radix sort, the authors suggest a merge-sort algorithm for the GPU, in which the a pair-wise merge tree is required in the final stages. In [9], a hybrid sorting algorithm is presented for the GPU. Initially the data is sorted using bucket sort and this is followed by a merge sort. The bucket approach suffers from workload imbalance and requires atomic instructions (i.e., synchronization).

Another focus of sorting algorithms is finding a way to implement them in a cache oblivious [10] way. As the algorithm in this paper focused on the merging stage and not the entire sort and presented a cache aware merging algorithm, we will not elaborate on cache oblivious algorithms. The interested reader is referred to [11-13].

VI. Implementation and Measurement Results

It is quite evident from the previous sections that we have succeeded in truly parallelizing the entire merging and sorting process, with negligible overhead for any numbers of interest. Nonetheless, we wanted to obtain actual performance results on real systems in order to ensure that we did not miss important issues.

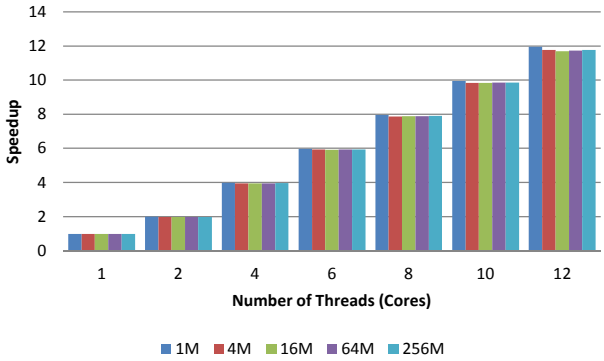

Figure 5 - Speedup of the regular Merge Path algorithm. Each of the colored bars represents a different sized input array. The sizes of the arrays are in Mega elements.

We implemented our basic Parallel Merge algorithm on a dual 6-core processor Intel x86 system. We begin with a brief overview of the system, including system specifications, and then present some of the practical challenges of implementing the algorithms. Following this, we present the speedup of the new algorithm. The runtime of Merge-Path with a single thread is used as the baseline.

We used a 2-processor, 2X6 core Intel X86 system with hyperthreading, It has L1 and L2 private caches for each core. The cores share an L3 cache. Because the cores have private caches, a cache coherency mechanism is required to ensure correctness. Furthermore, as we had multiple processors, each with its own L3 cache, the cache coherence mechanism had to communicate across processors; this is even more expensive from a latency point of view.

Specifically, we used a Dell-T610 server. The server consists of two X5670 INTEL processors, each of which having six cores with a private 32KB L1 data cache and a private 256KB L2 cache. Each processor has a 12MB L3 cache. The processors are connected via 6.4GT/s QPI. The server has 12GB DDR3 memory. For testing the algorithm, the following capabilities have been disabled: 1) INTEL hyper threading technology. 2) INTEL turbo technology. The reasons are fairly obvious.

Our implementation of Merge Path uses OpenMP. We tested the algorithm using multiple sizes of integer arrays and different numbers of threads. In view of the sophisticated cache management and prefetching of this system, we left this issue to the hardware and implemented the basic version of our algorithm rather than the segmented one. In Figure 5, the data set sizes refer to the size of each of the input arrays A and B. The output array S is twice this size, meaning that the total memory required for the 3 arrays is $4 \cdot |A| \cdot |type|$, where $|type|$ denotes the number of bytes need to stored the data type (for 32 bit integers this will be 4).

In Figure 5, we present the speedup of executing Merge Path using various size input arrays. One mega element

refers to 2^{20} elements. As can be seen, the speedups are near linear, with a slight reduction in performance for the bigger input arrays: approximately $11.7X$ for 12 threads.

Remark. We note that the single-thread execution time of our algorithm was some 6% longer than a truly sequential merge algorithm. This is due in part to a few extra instructions, and possibly also to overhead of OpenMP.

Both the basic and the segmented algorithm were also implemented on a semi-stable prototype of Hypercore [16], a many-core architecture with shared L1 cache that is effectively a CREW PRAM architecture and supports fine-grain task-level parallelism. These results confirmed our expectations, but we were unable to obtain end-to-end results due to an incomplete implementation of the cache system in that prototype.

VII. Conclusions

In this paper, we explored the issue of parallel sorting through the cornerstone of many sorting algorithms – the merging of two sorted arrays.

One important contribution of this paper is a very intuitive, simple and efficient approach to correctly partitioning each of two input sorted arrays into segments that, once pairs of segments, one from each, are merged, the concatenation of the merged pairs yields a single sorted array. This partitioning is also done in parallel.

Another important contribution is an insightful consideration of cache related issues. This are extremely important because, especially when parallelized, sorting and merging are carried out at a speed that is very often determined by the memory subsystem rather than by the compute power.

We implemented the algorithms on a multi-processor, multi-core X86 platform that represents mainstream computers. The results show that even though the algorithm was initially aimed at PRAM architectures the algorithm gives optimal speedups for the X86. This notwithstanding, sorting can be carried out in a much more cost- and power-efficient manner on many-core systems with lightweight compute cores. To this end, the efficient segmented version of our algorithm is very promising, as it can operate efficiently with simple caches.

Acknowlegements. The authors thank Rob McColl of the HPC lab at Georgia Institute of Technology for his suggestions and lengthy discussions on MergePath; Peleg Aviely and Shachar Raindel for their useful comments.

References

[1] T. H. Cormen, *et al.*, *Introduction to algorithms*. Cambridge, Mass.New York: MIT Press, 1990.

[2] N. Deo and D. Sarkar, "Parallel algorithms for merging and sorting," *Information Sciences,* vol. 56, pp. 151-161, 1991.

[3] J. L. Hennessy, *et al.*, *Computer architecture : a quantitative approach*, 4th ed. Morgan Kaufmann, 2007.

[4] K. E. Batcher, "Sorting Networks and Their Applications," Proc. AFIPS Conf. 1968.

[5] S. G. Akl and N. Santoro, "Optimal Parallel Merging and Sorting Without Memory Conflicts," *IEEE Trans. Computers,* vol. C-36, pp. 1367-1369, 1987.

[6] Y. Shiloach and U. Vishkin, "Finding the maximum, merging, and sorting in a parallel computation model," *J. Algorithms,* vol. 2, pp. 88-102, 1981.

[7] N. Deo, *et al.*, "An optimal parallel algorithm for merging using multiselection," *Information Processing Letters,* vol. 50, pp. 81-87, 1994.

[8] N. Satish, *et al.*, "Designing efficient sorting algorithms for manycore GPUs," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009, pp. 1-10.

[9] E. Sintorn and U. Assarsson, "Fast parallel GPU-sorting using a hybrid algorithm," *J. Parallel and Distributed Computing,* vol. 68, pp. 1381-1388, 2008.

[10] A. Aggarwal and S. V. Jeffrey, "The input/output complexity of sorting and related problems," *Commun. ACM,* vol. 31, pp. 1116-1127, 1988.

[11] R. A. Chowdhury, *et al.*, "Oblivious algorithms for multicores and network of processors," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010, pp. 1-12.

[12] R. Cole and V. Ramachandran, "Resource Oblivious Sorting on Multicores Automata, Languages and Programming." vol. 6198, S. Abramsky, *et al.*, Eds., ed: Springer Berlin / Heidelberg, 2010, pp. 226-237.

[13] M. Frigo, *et al.*, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*, 1999, pp. 285-297.

[14] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," presented at the Proceedings of the April 18-20, 1967, spring joint computer conference, Atlantic City, New Jersey, 1967.

[15] J. L. Gustafson, "Reevaluating Amdahl Law," *Commun. ACM,* vol. 31, pp. 532-533, May 1988.

[16] "HyperCore Software Developer's Handbook," ed: Plurality, 2009.