Visualizing PRAM Algorithm for Mergesort

Cade Wiley
Department of Computer Science
Wake Forest University
Winston-Salem, NC, USA
wilecr18@wfu.edu

Grey Ballard
Department of Computer Science
Wake Forest University
Winston-Salem, NC, USA
ballard@wfu.edu

***Abstract*—Undergraduate algorithms courses are a natural setting for teaching many of the theoretical ideas of parallel computing. Mergesort is a fundamental sequential divide-and-conquer algorithm often analyzed in such courses. In this work, we present a visualization tool to help demonstrate a novel PRAM algorithm for mergesort that is work efficient and has polylogarithmic span. Our implementation uses the Thread-Safe Graphics Library, which has an existing visualization of parallel mergesort. We demonstrate that our proposed algorithm has better work and span than the one currently visualized.**

***Index Terms*—multithreaded programming, animation, sorting, work/span analysis, OpenMP, pedagogical tools**

I. Introduction

Parallel computing topics can naturally be incorporated into undergraduate algorithms courses. In particular, introducing the PRAM model along with work/span analysis requires little overhead, and many of the algorithmic techniques, such as divide and conquer and dynamic programming, can be revisited in a parallel context. One of the popular textbooks for such courses devotes a chapter to multithreaded programming using PRAM with a focus on the divide-and-conquer algorithms for matrix multiplication and mergesort [1, Chapter 27].

We propose a visualization tool to help teach students a parallel algorithm for mergesort that is work efficient and has polylogarithmic span. The PRAM algorithm we propose in § II is different from the textbook algorithm [1]. While it has the same work and span complexity, we argue it simplifies the analysis. Our visualization tool is built using the Thread-Safe Graphics Library (TSGL) [2]–[4], which is designed to help students interact with and understand the execution of parallel programs. The library has an existing visualization of mergesort, but we demonstrate that the algorithm used is both work inefficient and has limited parallelism. We present screen captures of the animations of both existing and new algorithms in § III to illustrate the differences in the visualizations.

Combined with previous work on visualization of two parallel dynamic programming algorithms (Knapsack and Longest Common Subsequence) [5], the proposed tool can be used by instructors to demonstrate parallelization of two separate algorithmic paradigms using the PRAM model. We recommend the tool as a supplement to explanations of the parallel algorithm and analysis. Its interactivity can generate questions and suggestions from students in class, and it can be compared against algorithms with limited parallelism. As we describe in § IV, we hope to incorporate our new mergesort visualization into TSGL for wider dissemination. In this way, it will inherit the usability, documentation, and reach of the greater library.

This work is supported by the National Science Foundation under Grant Numbers CCF-1942892 and OAC-2106920.

II. PRAM Algorithms for Mergesort

Mergesort is a divide-and-conquer algorithm that yields a nice example of parellelization using the **spawn/sync** mechanism of the multithreaded programming paradigm [1, Section 27.3]. The recursive calls to sort each half of the input array are independent and can be executed in parallel. The merge operation cannot occur until both recursive calls have returned, so it executes after a synchronization. Algorithm 1 presents the pseudocode for parallel mergesort using a function call for the merge operation. We discuss multiple merge algorithms in the following subsections.

The mergesort pseudocode is written in C-style, with the input data initially copied into both arrays A and B and the sorted output stored in B upon return. Note that the recursive calls swap the source and target arrays, which is a pointer-swapping technique used to avoid an explicit copy. In this way, the merging is performed back and forth between the two arrays in memory across the levels of recursion. To ensure correctness for recursion depth of any parity, the function requires that the input data is initially stored in both arrays. We assume the merge function takes two sorted arrays as input and outputs the sorted union in the third argument.

Algorithm 1 PRAM Algorithm for Mergesort

Require: On input, A, B contain identical unsorted data
Ensure: On output, B is sorted in increasing order
1: **function** ParMergeSort(A, B)
2: **if** length$(A) = 1$ **then return**
3: Partition $A = \begin{bmatrix} A_L & A_R \end{bmatrix}$, $B = \begin{bmatrix} B_L & B_R \end{bmatrix}$ evenly
4: **spawn** ParMergeSort(B_L, A_L)
5: ParMergeSort(B_R, A_R)
6: **sync**
7: ParMerge(A_L, A_R, B)
8: **end function**

In the work/span analysis of parallel mergesort, we see that the complexity in both cases depends on the complexity of the

merge. The work $T_1^{\text{MS}}(n)$ and span $T_\infty^{\text{MS}}(n)$ of mergesort are given by the recurrences

$$\begin{aligned} T_1^{\text{MS}}(n) &= 2 \cdot T_1^{\text{MS}}(n/2) + T_1^{\text{M}}(n), \\ T_\infty^{\text{MS}}(n) &= T_\infty^{\text{MS}}(n/2) + T_\infty^{\text{M}}(n) \end{aligned} \tag{1}$$

where $T_1^{\text{M}}(n)$ and $T_\infty^{\text{M}}(n)$ are the work and span of merge, respectively. As long as $T_1^{\text{M}}(n) = O(n)$, we have that $T_1^{\text{MS}}(n) = O(n \log n)$, which implies that parallel mergesort is work efficient. The span of merge varies based on how merge is parallelized; we consider four alternatives below.

A. Sequential Merge

We first consider performing a sequential merge. In this case, we have identical work and span: $T_1^{\text{M}}(n) = T_\infty^{\text{M}}(n) = O(n)$. This implies from eq. (1) that the span of mergesort is $T_\infty^{\text{MS}}(n) = O(n)$, as it is dominated by the final merge.

While this approach yields a work-efficient mergesort, this span implies that the parallelism, or limit of perfect scaling, is $O(\log n)$, which is quite small. To obtain efficient span, we must parallelize the merge operation.

B. Sequential In-Place Merge

The mergesort algorithm currently visualized within TSGL (see § III-B) uses a sequential *in-place* merge. That is, it does not require the extra memory of a temporary buffer. The merge procedure is performed using a circular shift of input data (when necessary). That is, the algorithm processes items from the two consecutive sorted arrays, and when the left array's item is smaller, no work is required. When the right array's item is smaller, that item is shifted into place and the larger items of the left array are all shifted to the right. Because the circular shift requires $O(n)$ operations, and it occurs $O(n)$ times in the worst case, this in-place merge costs $O(n^2)$.

The in-place merge is sequential, so we have identical work and span: $T_1^{\text{M}}(n) = T_\infty^{\text{M}}(n) = O(n^2)$. This implies from eq. (1) that the work and span of mergesort is dominated by the final merge: $T_1^{\text{MS}}(n) = T_\infty^{\text{MS}}(n) = O(n^2)$. Thus, the algorithm is not work efficient and has constant parallelism.

C. Parallel Merge using Median of One Input

The first parallel merge we consider is proposed in [1, Chapter 27]. The idea is to determine a value by which we can partition the output array into a left part and a right part. Because the input arrays are sorted, we can also partition those arrays by the value. In this way, the left parts of the input can be merged into the left part of the output independently from the merge of the right parts. We can select the median of the larger array in constant time and then use (sequential) binary search to partition the smaller array by that value. After determining the partitioning, we use two recursive calls to merge both parts in parallel. Choosing the median of the larger array guarantees that the size of the larger part of the output is no more than 3/4 the size of the entire array.

We first consider the span of this parallel merge algorithm. Because the left and right merges execute in parallel, the span recurrence is based on the more costly of the two parts, which is bounded above in size by $3n/4$. Thus, the span recurrence is given by $T_\infty^{\text{M}}(n) \leq T_\infty^{\text{M}}(3n/4) + O(\log n)$ and solves to $T_\infty^{\text{M}}(n) = O(\log^2 n)$. The work recurrence is given by

$$T_1^{\text{M}}(n) = T_1^{\text{M}}(\alpha n) + T_1^{\text{M}}((1-\alpha)n) + O(\log n)$$

where $1/4 \leq \alpha \leq 3/4$ and α can vary throughout the recurrence. This recurrence solves to $T_1^{\text{M}}(n) = O(n)$, so that we see this recursive approach is work efficient, though the recurrence is harder to solve.

Given this analysis of the merge algorithm and eq. (1), we see that using this as a subroutine within the mergesort algorithm obtains a mergesort span of $O(\log^3 n)$. With poly-logarithmic span, we obtain much larger parallelism.

Algorithm 2 PRAM Median-of-Union Algorithm for Merge

Require: On input, A and B are individually sorted arrays
Ensure: On output, S is sorted union of A and B

1: **function** ParMerge(A, B, S)
2:     **if** length(A) $= 0$ **and** length(B) $= 0$ **then return**
3:     Find median m of $A \cup B$
4:     Partition $A = [A_L \quad A_R]$, $B = [B_L \quad B_R]$ by m
5:     Partition $S = [S_L \quad S_R]$ evenly
6:     **spawn** ParMerge(A_L, B_L, S_L)
7:             ParMerge(A_R, B_R, S_R)
8:     **sync**
9: **end function**

D. Parallel Merge using Median of Union of Inputs

We propose an alternative parallel merge algorithm based on computing the median of the union of the two input arrays, given in Alg. 2 and illustrated in Fig. 1. It works by partitioning the input arrays by the median of their union so that the output array can be partitioned evenly. Again, we use C-style pseudocode with the output array S passed in as a parameter and overwritten with the sorted data. The main advantage of this approach is that the even division of the output array yields work and span recurrences that are easier to solve than the parallel merge algorithm described in § II-C. Computing the median of the union of two sorted arrays is more complicated

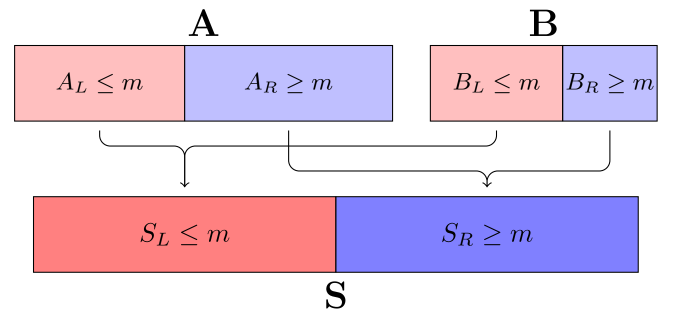


Fig. 1: Diagram of arrays for recursive merge (Alg. 2) using median of union with value m so that the output array is evenly divided for recursive calls, denoted by red and blue colors.

than a single binary search, but it has complexity $O(\log n)$ and is a nice homework problem for students learning the (sequential) divide-and-conquer algorithmic technique (see [1, Exercise 9.3-8] and [6, Exercise 2.22]).

Given the complexity of the sequential median-of-union algorithm, the work and span recurrences are given by

$$T_1^{\mathrm{M}}(n) = 2T_1^{\mathrm{M}}(n/2) + O(\log n)$$
$$T_\infty^{\mathrm{M}}(n) = T_\infty^{\mathrm{M}}(n/2) + O(\log n),$$

which solve to $T_1^{\mathrm{M}}(n) = O(n)$ and $T_\infty^{\mathrm{M}}(n) = O(\log^2 n)$. We note that the work recurrence is solved via a straightforward application of the master theorem. From eq. (1), this implies that the work of mergesort is $T_1^{\mathrm{MS}}(n) = O(n \log n)$ (so it is work efficient) and the span of mergesort is $T_\infty^{\mathrm{MS}}(n) = O(\log^3)$, which matches the complexities of the median-of-one-input algorithm described in § II-C.

III. Mergesort Visualizations

In this section we describe the existing mergesort visualization as well as our proposed implementation and visualization. Both use the Thread-Safe Graphics Library (TSGL), described in more detail in § III-A, and both implementations use OpenMP. The current implementation is iterative and works bottom-up, with a single parallel region and manual task assignment. Our implementation is recursive and uses nested task parallelism, so its structure follows Algs. 1 and 2 more closely. We build upon the existing implementation, following many of its visualization conventions and color choices.

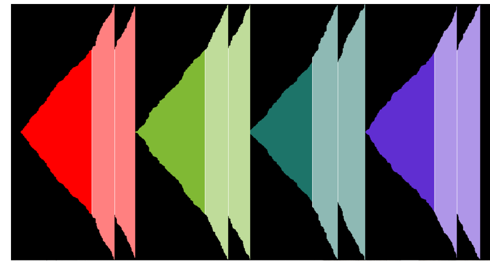
We distinguish between two phases of computation within parallel mergesort to compare the current visualization with our own. Letting n be the length of the array and p be the number of threads, we define the first phase to be the execution of the bottom $\log(n/p)$ recursive levels, which corresponds to each thread independently sorting a subarray of approximate length n/p. We define the second phase to be the execution of the top $\log p$ levels, which corresponds to the merging of p individually sorted subarrays.

A. TSGL Background

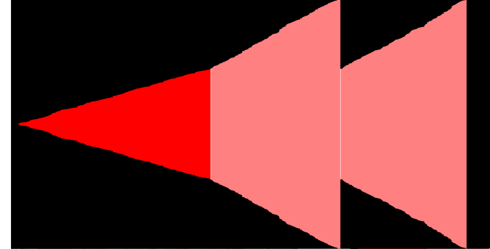
TSGL is a C++ library that allows for real time visualizations of multithreaded programs [2]–[4]. Currently TSGL contains about 30 example visualizations including image processing, numerical methods, sorting, dynamic programming, and more. The innovation of TSGL is that it allows thread-safe drawing to a shared canvas during parallel computation. It is designed for primarily pedagogical purposes, allowing students to visualize the behavior of multiple threads working simultaneously. As such, it comes with the ability to pause the canvas to slow down the action for users with sleep functions.

B. TSGL's Current Visualization

The current TSGL visualization demonstrates parallel mergesort using a sequential in-place merge. See § II-B for the work/span analysis of this algorithm. The visualization itself, as shown in Fig. 2, is a single array with entries represented by rectangles stacked side by side and values that correspond to rectangle height. The rectangles progressively get sorted from shortest on the left to tallest on the right. White is used to signify active progress on a rectangle and unique colors represent the work done by each processor. An opaque color demonstrates that the processor has already performed its task upon the data and a translucent color shows that the processor has not yet performed its task upon the data.



(a) Phase 1: independent mergesorts in parallel



(b) Phase 2: sequential final merge

Fig. 2: TSGL's current parallel mergesort visualization.

1) Phase 1: Phase 1 is the independent sorting of subarrays of size approximately n/p. A screen capture of the first phase running with four processors is shown in Fig. 2a. In this example, all four processors are working independently on each subarray, and we see that each processor is performing the final merge for the subarray. In this phase, the algorithm is achieving full parallelism and is perfectly load balanced. In the animation, the left translucent input array of each merge shifts to the right as the merge operation proceeds, as the algorithm works in place. As mentioned in § II-B, when the right subarray of a merge has a value smaller than the left subarray, the larger values of the left array must be shifted. This subarray shift occurs within a single visualization step, so the $O(n)$ cost is hidden from the viewer.

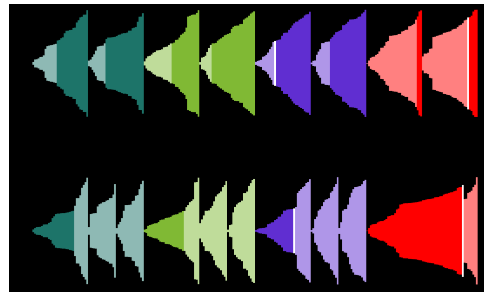
2) Phase 2: After each processor has sorted their subarray of approximately n/p data, the p sorted subarrays must be merged. During this second phase, the use of a sequential merge implies that the algorithm loses parallelism as processors start to drop out of the computation. In the example shown in Fig. 2, after Phase 1 completes, only two processors are active, each merging 2 of the remaining 4 subarrays. Figure 2b depicts the final merge of two sorted subarrays into the final output. In this case, only one processor is active, which is why we observe only one color, and it is performing $O(n^2)$ work. This loss of parallelism leads to a large span and limited theoretical possible speedup.

C. Our Visualization

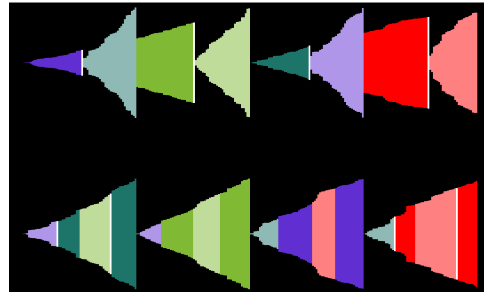
We implement our proposed algorithm as described in § II-D using many of the conventions of TSGL's current visualization. Because our algorithm is out of place, we visualize both arrays, and we display the sorted output in the second (bottom) array. We again use rectangle height to denote value, the same color scheme to distinguish among processors, and white to signify active progress. One difference in our visualization is that we shade a rectangle with a translucent color whenever a particular processor reads the corresponding value and with an opaque color whenever a processor writes the corresponding value. Screen captures of our visualization are given in Fig. 3.

1) Phase 1: The first phase of our visualization is similar to TSGL's current visualization where each processor independently sorts their subarray. In the animation, we can observe the merging processing working both from top to bottom and bottom to top as the pointers get swapped in the depth of the recursion. Fig. 3a shows the very end of Phase 1 for four processors, and we see that it looks similar to TSGL's current visualization (see Fig. 2a) except for the fact that it uses a second array. Note that the color order is different because the OpenMP scheduler nondeterministically maps tasks to threads in our implementation.

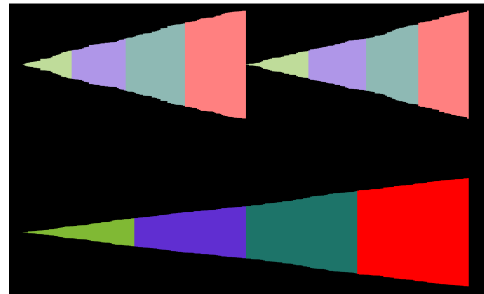
2) Phase 2: The second phase of our visualization differs greatly from the current TSGL visualization, as we use a parallel merge algorithm as described in § II-D. In our implementation, all processors work throughout Phase 2. Figure 3b depicts the merging of four sorted subarrays into two subarrays (note that at this depth, the merging occurs from bottom to top). In this screen capture, we see that each of the processors is merging into a subarray of the top array of size approximately $n/4$, but we note that the two input subarrays in the bottom array have varying sizes. We also show the end of Phase 2 in Fig. 3c to see that in the final merge of two sorted subarrays, all four processors write approximately $n/4$ data to the bottom array. This load balancing allows for low span and high theoretical parallel speedup.



(a) Phase 1: independent mergesorts in parallel



(b) Phase 2: full parallelism across two merges



(c) Phase 2 complete: end of final merge in parallel

Fig. 3: Our parallel mergesort visualization.

IV. Conclusion

Parallel mergesort is an ideal topic for incorporating parallel computing into an algorithms course. Students can recognize the independence between recursive calls for natural task parallelism, but parallelizing the merge is a challenging task. Our proposed algorithm is work efficient and has polylogarithmic span, and it utilizes a clever algorithm for computing the median of the union of two sorted arrays. Our visualization helps students to understand the execution of the recursive structure and observe the importance of merging in parallel.

The tool is currently being used for in-class demonstrations in multiple algorithms courses at Wake Forest. Students are encouraged to suggest input parameters to illustrate both the flexibility of the algorithm as well as the limitations. For example, students can observe load imbalance for non-powers-of-two numbers of processors. In follow-up assignments, students can be asked to fill in gaps in the work/span analysis, design PRAM algorithms for problems with similar structure such as the FFT, or implement mergesort using OpenMP task-based parallelism. We plan to incorporate our visualization into the main TSGL repository so that it can be used by more instructors.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[2] J. C. Adams, P. A. Crain, and M. B. Vander Stel, "TSGL: A thread safe graphics library for visualizing parallelism," *Procedia Computer Science*, vol. 51, pp. 1986–1995, 2015. [Online]. Available: https://doi.org/10.1016/j.procs.2015.05.463

[3] J. C. Adams, P. A. Crain, and C. P. Dilley, "Seeing multithreaded behavior using TSGL," in *Proceedings of EduPar*, 2016, pp. 972–977. [Online]. Available: https://doi.org/10.1109/IPDPSW.2016.17

[4] J. C. Adams *et al.*, "TSGL: A tool for visualizing multithreaded behavior," *Journal of Parallel and Distributed Computing*, vol. 118, pp. 233–246, 2018. [Online]. Available: https://doi.org/10.1016/j.jpdc.2018.02.025

[5] G. Ballard and S. Parsons, "Visualizing parallel dynamic programming using the thread safe graphics library," in *Proceedings of EduHPC*, 2021, pp. 24–31. [Online]. Available: https://doi.org/10.1109/EduHPC54835.2021.00009

[6] S. Dasgupta, C. H. Papadimitriou, and U. Vazirani, *Algorithms*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 2006.