Parallel Divide-and-Conquer Algorithms for Bubble Sort, Selection Sort, and Insertion Sort

Pramod Ganapathi Rezaul Chowdhury

State University of New York at Stony Brook, New York, USA
Email: {pramod.ganapathi,rezaul}@cs.stonybrook.edu

We present efficient parallel recursive divide-and-conquer algorithms for bubble sort, selection sort, and insertion sort. Our algorithms have excellent data locality and are highly parallel. The computational complexity of our insertion sort is $\mathcal{O}\left(n^{\log_2 3}\right)$ in contrast to $\mathcal{O}\left(n^2\right)$ of standard insertion sort.

Keywords: parallel divide-and-conquer; sorting algorithm; bubble sort; selection sort; insertion sort; merge sort; quicksort

1. INTRODUCTION

Sorting is a computational process of rearranging a multiset of items in non-descending or non-ascending order [1]. Sorting is used in real-life scenarios. Smartphone contacts are sorted based on names; students' (resp. employees' and patients') profiles are sorted based on student ID (resp. employee ID and patient ID); flight (or bus or train) information is sorted based on time-of-departure; and finally, the importance given to different jobs/people are sorted based on our priorities.

Sorting is one of the most fundamental problems in computer science. Sorting is used as an intermediate step to solve several computer science problems [1, 2] such as: bringing all items with the same identification together, matching items in two or more files, searching for a specific value, testing if all elements are unique, deleting duplicates, finding the kth most frequently occurring element, finding set union/intersection, finding the closest pair of points, finding a convex hull, and so on. Even if sorting was totally useless, it is an exceptionally interesting problem that leads to several beautiful algorithms and analyses. For all these reasons, sorting is a well-studied problem and has a large literature.

Several algorithms have been discovered to sort an array of size n. More than six decades of research has yielded over a hundred algorithms to solve the sorting problem, many of which are either minor or major variations of tens of standard algorithms. Sorting algorithms can be classified based on a wide variety of conditions such as computational complexity, in-place or not-in-place, stable or unstable, recursive or non-recursive, comparison-based or not comparison-based, deterministic or probabilistic, internal-memory or external-memory, serial or parallel [3, 4], shared-memory or distributed-memory, adaptive [5] or non-adaptive, and self-improving [6] or non-self-improving.

Divide-and-Conquer Algorithms for Bubble, Selection, and Insertion Sorts. Bubble, selection, and insertion sorts [7] are some of the most elementary sorting algorithms that are widely taught to the students of computer science in the lower-level undergraduate courses on algorithms and/or data structures. These algorithms are inefficient but popular majorly because they are arguably simple, intuitive, easy-to-remember, and easy-to-program. Some of the fastest sorting algorithms such as merge sort and quicksort are then introduced to teach the power of the divide-and-conquer algorithm design technique.

Divide-and-conquer (D&C) is a powerful algorithm design technique used to solve a problem by dividing it into two or more subproblems, solving them, and combining their solutions to solve the original problem. D&C algorithms have the following important advantages. (i) They are sometimes *efficient* [7] in the sense that they reduce the total number of computations. (ii) They often are *cache-efficient*[1] [8, 9, 10, 11], *cache-oblivious*[2] [12], and *cache-adaptive*[3] [13, 14].

There are several other benefits of using the D&C design technique. (i) The complexities (e.g.: time, space, and communication) of D&C algorithms can be analyzed using *recurrences* [15] as such algorithms are typ-

[1] *Cache-efficient algorithms* are those that make efficient use of caches by bringing in as less data as possible and using them as much as possible before evicting them.

[2] *Cache-oblivious algorithms* are those that don't need to know machine parameters such as cache sizes, block sizes, and the number of levels of memory. Such algorithms are portable across machines with different cache parameters.

[3] *Cache-adaptive algorithms* are those that can adapt well to computing systems in which the available memory changes dynamically.

ically implemented recursively. (*ii*) They can be *parallelized easily* [16] as the subproblems are typically independent. (*iii*) There exist frameworks to *automatically or semi-automatically generate* D&C algorithms for certain classes of computational problems [10, 17, 18, 19]. (*iv*) The D&C algorithms can be *architecture-independent*[4] [20].

So, a natural question to ask is:

> **Question**
>
> Is it possible to design parallel divide-and-conquer algorithms for bubble, selection, and insertion sorts to improve computational complexity and/or data locality?

In this paper, we answer the question above affirmatively by designing cache-efficient parallel D&C algorithms for bubble sort, selection sort, and insertion sort. Table 1 shows performance comparison of our algorithms with the existing algorithms. All our algorithms incur asymptotically fewer cache misses than those of their iterative counterparts because they exploit *temporal data locality*[5]. Our algorithms are also highly parallel. Furthermore, our insertion sort performs $\mathcal{O}\left(n^{\log_2 3}\right)$ computations, i.e., ***polynomially better*** than that of standard insertion sort. But, it is also true that asymptotically-improved bubble, selection, and insertion sorts are no match to the world's fastest sorting algorithms. Nevertheless, the most important usefulness of our algorithms is not performance but ***pedagogy***. In addition, designing and analyzing such algorithms is ***theoretically interesting***. The presented algorithms serve as good examples or exercises to teach the design and analysis of ***parallel divide-and-conquer algorithms***.

A parallel comparison sorting algorithm is *work-optimal* if it performs $\Theta\left(n \log n\right)$ computations. Some parallel D&C sorting algorithms such as 2-way merge sort [21], 2-way randomized quicksort [21], $\sqrt{n}$-way randomized sample sort [21], $\sqrt[3]{n}$-way funnelsort [12], and $\sqrt{n}$-way distribution sort [12] are work-optimal and the computational complexity for randomized quicksort is *whp*[6]. Sorting algorithms such as ours are not work-optimal.

There are similarities and differences between our sorting algorithms and parallel merge sort and parallel randomized quicksort. Our bubble and selection sorts use algorithm-specific Partition idea similar to that of quicksort. Likewise, our insertion sort uses algorithm-specific Merge idea similar to that of merge sort. Unlike merge sort and quicksort, our algorithms are not computationally efficient. This is because of the slow Partition and Merge functions that our algorithms use, which call themselves four or three times, respectively. In contrast, the Merge function of merge sort calls itself twice [21]. Similarly, the Partition function of quicksort usually uses the PrefixSum function [21] instead of calling Partition recursively and the PrefixSum function can be considered as a 2-way D&C. Some more differences are as follows. The Merge function in standard merge sort is not-in-place whereas the Merge function in our D&C insertion sort is in-place. Quicksort Partition works on an array and uses a randomized pivot whereas the Partition function in our D&C bubble and selection sorts works on two arrays and does not use a pivot.

Our Contributions. The major contributions of this paper are:
(1) [**Theory**.] We present parallel recursive divide-and-conquer algorithms for bubble sort, selection sort, and insertion sort. We prove their correctness and analyze their complexities, as shown in Table 1. Our insertion sort performs $\mathcal{O}\left(n^{\log_2 3}\right)$ computations, i.e., polynomially better than that of standard iterative insertion sort. Our algorithms are cache-efficient and highly parallel.
(2) [**Practice**.] We implement the parallelized and optimized versions of our algorithms and compare them with their standard iterative counterparts to achieve around 20× to 1300× speedup, depending on the algorithm and the input distribution.

Related Work. Variations of bubble sort [22, 23] exist such as cocktail sort [1], where the direction of bubbling alternates between left-to-right and right-to-left, and odd-even sort [24]. Selection sort variants include bingo sort [25], which scans the remaining elements to find the greatest value and shifts all elements with that value to their final locations.

Insertion sort variants include Shell sort [26], where elements separated by a distance are compared; binary insertion sort, which uses binary search to find the exact location of the new elements to be inserted; heap sort [27], where insertions and searches are performed with a sophisticated data structure called a heap; and library sort [28], where small number of spaces are left unused to make gaps for the elements to be inserted. Insertion sort is generally faster than selection sort which typically is faster than bubble sort.

The cache performance of sorting algorithms have been studied by LaMarca and Ladner [29]. The lower-bounds for data transfers for external-memory sorting algorithms are given by Aggarwal and Vitter [30].

Model of Computation. We analyze the performance of a parallel algorithm on a shared-memory multicore machine in the *binary-forking model* [31] using

[4] *Architecture-independent algorithms* are those that run on both shared-memory and distributed-memory machines with almost no change to their core algorithmic structure.

[5] An algorithm must have the following two features in order to make good use of cache. (*i*) *Spatial data locality*: Whenever a cache block is brought into the cache, it contains as much useful data as possible. (*ii*) *Temporal data locality*: Whenever a cache block is brought into the cache, as much useful work as possible is performed on this data before removing the block from the cache.

[6] An event E is said to occur *with high probability (whp)* if Probability$(E) \geq 1 - \alpha/n^{\beta}$, where $\alpha \geq 1$ and $\beta > 0$ are constants.

Problem	Existing Algorithm Work (T_1)	Existing Algorithm Serial cache complexity (Q_1)	Existing Algorithm Span (T_∞)	Our Divide-and-Conquer Algorithm Work (T_1)	Our Divide-and-Conquer Algorithm Serial cache complexity (Q_1)	Our Divide-and-Conquer Algorithm Span (T_∞)	Result
Bubble sort [7]	$\Theta(n^2)$	$\Theta\left(\frac{n^2}{B}\right)$	$\Theta(n^2)$	$\Theta(n^2)$	$\mathcal{O}\left(\frac{n^2}{BM}\right)$	$\Theta(n)$	Section 2
Selection sort [7]	$\Theta(n^2)$	$\Theta\left(\frac{n^2}{B}\right)$	$\Theta(n^2)$	$\Theta(n^2)$	$\mathcal{O}\left(\frac{n^2}{BM}\right)$	$\Theta(n)$	Section 3
Insertion sort [7]	$\mathcal{O}(n^2)$	$\mathcal{O}\left(\frac{n^2}{B}\right)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{\log_2 3})$	$\mathcal{O}\left(\frac{n^{\log_2 3}}{BM^{\log_2 3-1}}\right)$	$\mathcal{O}(n)$	Section 4

TABLE 1. Work (T_1), serial cache complexity (Q_1), and span (T_∞) of iterative and recursive divide-and-conquer algorithms for bubble sort, selection sort, and insertion sort. The serial cache complexity is given for $n = \Omega(M)$, where $M \geq B$. All notations used are described in Table 2.

the *work-span* performance metrics [32]. The total number of computations of a parallel algorithm is called *work* and denoted by $T_1(n)$. It is also the serial running time of the algorithm. The running time of an algorithm on a machine with an infinite number of processors is called *span* and denoted by $T_\infty(n)$. The *parallel running time* of an algorithm on p processors, denoted by $T_p(n)$, when scheduled by a greedy scheduler is given by $T_p(n) = \mathcal{O}(T_1(n)/p + T_\infty(n))$. The *parallelism* of an algorithm is computed by the ratio $T_1(n)/T_\infty(n)$.

We analyze the performance of an algorithm using the *cache-oblivious model* (or *ideal-cache model*) [12]. We measure the total number of cache misses or page faults, called *cache complexity.* Cache complexity captures the total number of data transfers between adjacent levels of memory. It is important to reduce data movements because communication is usually more expensive than computation. The *serial cache complexity*, denoted by $Q_1(n)$, is the cache complexity of an algorithm on a serial machine. On the other hand, the *parallel cache complexity* is $Q_p(n) = \mathcal{O}(Q_1(n) + p(M/B)T_\infty(n))$ with high probability when run under the randomized work-stealing scheduler on a p-processor parallel machine with private caches. Here, M and B denote the cache size and the cache line size, respectively, where $M \geq B$.

Organization of the Paper. The paper is organized as follows. In Sections 2, 3, and 4, we present divide-and-conquer algorithms for bubble sort, selection sort, and insertion sort, respectively. In Section 5, we present experimental results of our algorithms. We conclude in Section 6.

2. BUBBLE SORT

Table 2 summarizes the notations used in the paper. The array to be sorted is $A[0..n-1]$. For simplicity, we assume that n is a power of 2. In all recursive function calls, we use notations such as $\ell, h, m, \ell\ell, \ell h, r\ell, rh$, etc, all of which represent indices in array A. Notations ℓ, m, h mean low, mid, and high, respectively. Terms $\ell\ell$ and rm mean low in the left subarray and mid in the right subarray, respectively. Other terms can be defined in a similar way. When subproblem size $(h - \ell + 1)$ or say $(\ell h - \ell\ell + 1)$ becomes less than or equal to the base case size b, then we execute an iterative base case kernel having an algorithm-dependent logic.

Symbol	Meaning
A	Array to be sorted
n	Number of array elements
b	Base case size
ℓ, h, m	Low, high, and mid
$\ell\ell$	Left subarray's low index
rh	Right subarray's high index
p	Number of processors
M, B	Cache size, cache line size
T_1	Work or total #computations
T_∞	Span or critical-path length
T_p	Parallel running time
T_1/T_∞	Parallelism
Q_1	Serial cache complexity
Q_p	Parallel cache complexity

TABLE 2. Notations used in paper.

A simple iterative algorithm BUBBLESORT-ITERATIVE is given in Figure 1. It has n iterations. In each iteration i ($\in [0, n-1)$), every two adjacent elements j and $j+1$, where $j \in [0, n-i-1]$, are compared and sorted if they are not already in their sorted order. The number of comparisons at iteration i is $n - i$ and at the end of the iteration, the array's $(i+1)$th largest element will be in its correct position.

A recursive divide-and-conquer bubble sort algorithm BUBBLESORT is shown in Figure 1. The aim is to sort the entire array $A[0..n-1]$. The function BUBBLESORT($A[\ell..h]$) sorts the subarray $A[\ell..h]$. The initial invocation to the algorithm is BUBBLESORT($A[0..n-1]$). The function in turn calls the PARTITION function. The PARTITION function brings the smallest $n/2$ elements to the left half and the largest $n/2$ elements to the right half of array A. Once the array A is partitioned, then BUBBLESORT is recursively called onto the left and right halves in parallel to sort the two halves. After the two halves are sorted recursively, the entire array $A[0..n-1]$ will be sorted. When a subproblem reaches the base case, it is sorted using the standard iterative bubble sort logic.

The partition function PARTITION($A[\ell\ell..\ell h]$, $A[r\ell..rh]$) partitions the elements such that after the partition, the largest element in $A[\ell\ell..\ell h]$ will be less than or equal to the smallest element in $A[r\ell..rh]$. The function works as follows.

```
BubbleSort-Iterative(A[0..n − 1])
for i ← 0 to n − 1 do
  for j ← 0 to n − i − 1 do
    if A[j] > A[j + 1] then
      Swap(A[j], A[j + 1])
```

```
BubbleSort(A[ℓ..h])
Sorts a given array by parallel divide-and-conquer bubble sort
Input: Subarray A[ℓ..h] of orderable elements
Output: A[ℓ..h] sorted in nondecreasing order
if (h − ℓ + 1) ≤ b then
  for i ← ℓ to h − 1 do
    for j ← ℓ to ℓ + h − i − 1 do
      if A[j] > A[j + 1] then
        Swap(A[j], A[j + 1])
else
  m ← (ℓ + h)/2
  Partition(A[ℓ..m], A[m + 1..h])
  par: BubbleSort(A[ℓ..m]),
       BubbleSort(A[m + 1..h])
```

```
Partition(A[ℓℓ..ℓh], A[rℓ..rh])
Partitions elements between two subarrays without using a pivot
Input: Two equal sized non-overlapping subarrays A[ℓℓ..ℓh] and A[rℓ..rh]
Output: Partition the elements such that all elements in A[ℓℓ..ℓh] are less than or equal to all elements in A[rℓ..rh]
1.  if (ℓh − ℓℓ + 1) ≤ b then
2.    for i ← rℓ to rh do
3.      for j ← ℓℓ to ℓh − 1 do
4.        if A[j] > A[j + 1] then
5.          Swap(A[j], A[j + 1])
6.      if A[ℓh] > A[i] then
7.        Swap(A[ℓh], A[i])
8.  else
9.    ℓm ← (ℓℓ + ℓh)/2; rm ← (rℓ + rh)/2
10.   par: Partition(A[ℓℓ..ℓm], A[rℓ..rm]),
           Partition(A[ℓm+1..ℓh], A[rm+1..rh])
11.   par: Partition(A[ℓℓ..ℓm], A[rm + 1..rh]),
           Partition(A[ℓm + 1..ℓh], A[rℓ..rm])
```

FIGURE 1. A recursive divide-and-conquer bubble sort algorithm. Initial call to the recursive algorithm is BUBBLESORT($A[0..n-1]$), where $A[0..n-1]$ is the array to be sorted.

In the base case, we use two loops: the outer-loop ranging over the right subarray and the inner-loop ranging over the left subarray. Using a logic similar to that of iterative bubble sort, the largest elements in the left subarray are pushed to the right subarray after every iteration.

In the recursion case, the PARTITION calls itself four times. The reason for requiring four function calls is simple. Let ℓm and rm be the midpoints of the left and right subarray, respectively. The left subarray $A[\ell\ell..\ell h]$ can be divided into two subarrays $A[\ell\ell..\ell m]$ and $A[(\ell m+1)..\ell h]$ and the right subarray $A[r\ell..rh]$ can be divided into two subarrays $A[r\ell..rm]$ and $A[(rm+1)..rh]$. This means there are a total of four possible combinations of left and right subarrays. In the first parallel step, the PARTITION function invokes two PARTITION functions in parallel that work on different regions of the array. In the second parallel step, two more PARTITION functions are invoked in parallel that work on disjoint regions. After the four self-invocations, the larger elements of the entire array would have moved to the right subarray leaving the smaller elements in the left subarray.

The proof of correctness and complexity analysis of BUBBLESORT are given in Theorems 2.1 and 2.2, respectively.

THEOREM 2.1 (**Bubble Sort Correctness**). BUBBLESORT *correctly sorts an unsorted array.*

Proof. We use mathematical induction to prove the theorem. First we prove the correctness of PARTITION function. Then we prove BUBBLESORT correct. We assume that n and b are powers of 2 such that $n \geq b$. We say $A[\ell\ell..\ell h]$ and $A[r\ell..rh]$ as left and right input subarrays, respectively.

(1) [Correctness of PARTITION.]
Basis. The base case logic when the input subarray is of size b is straightforward. The external loop runs b times and in each iteration, an element that is greater than or equal to b number of elements moves to the right subarray.
Induction. We assume that PARTITION works correctly when the input subarrays are of size 2^k for some k, such that $2^k \geq b$. We need to prove that PARTITION works for subarrays of size 2^{k+1}.

Let Q_1, Q_2, Q_3, and Q_4, where Q stands for "quarter", represent the subarrays $A[\ell\ell..\ell m], A[(\ell m+1)..\ell h], A[r\ell..rm]$, and $A[(rm+1)..rh]$, respectively, where each subarray is of size 2^k. Let W, X, Y, and Z be the initial sets (not lists) of numbers present at Q_1, Q_2, Q_3, and Q_4, respectively. Let SMALL(S_1, S_2) (resp. LARGE(S_1, S_2)) of two equal-sized sets S_1 and S_2 of numbers represent a set consisting of the smallest half (resp. largest half) of the numbers from sets S_1 and S_2. Also, let $S_1 \leq S_2$ denote that all elements of S_1 are less than or equal to all elements of S_2.

Consider the PARTITION function in Figure 1. After execution of line 9, the states of the four quarters of the array A are $Q_1 = W$, $Q_2 = X$, $Q_3 = Y$, and $Q_4 = Z$. After execution of line 10, the states of the four quarters of the array A are: $Q_1 =$ SMALL(W, Y), $Q_2 =$ SMALL(X, Z), $Q_3 =$ LARGE(W, Y), and $Q_4 =$ LARGE(X, Z). After execution of line 11, the states of the four quarters of the array A are:

$$Q_1 = \textsc{Small}(\textsc{Small}(W,Y), \textsc{Large}(X,Z))$$
$$Q_2 = \textsc{Small}(\textsc{Small}(X,Z), \textsc{Large}(W,Y))$$
$$Q_3 = \textsc{Large}(\textsc{Small}(X,Z), \textsc{Large}(W,Y))$$
$$Q_4 = \textsc{Large}(\textsc{Small}(W,Y), \textsc{Large}(X,Z))$$

It is easy to see that

$$Q_1 \le Q_4 \text{ and } Q_1 \le \textsc{Small}(W,Y) \le \textsc{Large}(W,Y) \le Q_3$$
$$Q_2 \le Q_3 \text{ and } Q_2 \le \textsc{Small}(X,Z) \le \textsc{Large}(X,Z) \le Q_4$$

As $Q_1 \le Q_3$, $Q_1 \le Q_4$, $Q_2 \le Q_3$, and $Q_2 \le Q_4$, the input subarrays of size 2^{k+1} have been partitioned.

(2) [Correctness of BubbleSort.]
Basis. The base case when the input subarray is of size b is exactly the same as the standard iterative bubble sort.
Induction. We assume that BubbleSort works correctly when the input subarrays are of size 2^k for some k, such that $2^k \ge b$. We need to prove that BubbleSort works for input subarrays of size 2^{k+1}. We know that the Partition function is correct and hence after line 8, the left subarray ($A[\ell..m]$) and the right subarray ($A[(m+1)..h]$) would be partitioned such that the largest element in the left subarray will not be greater than the smallest element in the right subarray. Then after line 9, we recursively sort the subarrays without affecting the partition constraint and hence the total subarray will be sorted. □

Theorem 2.2 (**Bubble Sort Complexity**). *BubbleSort incurs $\mathcal{O}(n^2/(BM) + n/B + 1)$ cache misses and has $\Theta(n)$ span.*

Proof. Let $Q_1^f(n)$ and $T_\infty^f(n)$ denote the number of serial cache misses and span of algorithm f, respectively. Let BS-I, BS, and Part denote BubbleSort-Iterative, BubbleSort, and Partition, respectively. Then,

$$Q_1^{\text{BS-I}}(n) = \textstyle\sum_{i=0}^{n-1} \Theta\left(((n-i)/B)+1\right) = \Theta\left(n^2/B + n\right).$$

$$\begin{array}{ll} Q_1^{\text{BS}}(n) = Q_1^{\text{Part}}(n) = \mathcal{O}\left(n/B+1\right) & \text{if } n \le \gamma M, \\ Q_1^{\text{BS}}(n) = 2Q_1^{\text{BS}}\left(n/2\right) + Q_1^{\text{Part}}\left(n/2\right) + \mathcal{O}\left(1\right) & \text{if } n > \gamma M, \\ Q_1^{\text{Part}}(n) = 4Q_1^{\text{Part}}\left(n/2\right) + \mathcal{O}\left(1\right) & \text{if } n > \gamma M. \end{array}$$

$$\begin{array}{ll} T_\infty^{\text{BS}}(n) = T_\infty^{\text{Part}}(n) = \mathcal{O}\left(1\right) & \text{if } n = 1, \\ T_\infty^{\text{BS}}(n) = T_\infty^{\text{BS}}\left(n/2\right) + T_\infty^{\text{Part}}\left(n/2\right) + \mathcal{O}\left(1\right) & \text{if } n > 1, \\ T_\infty^{\text{Part}}(n) = 2T_\infty^{\text{Part}}\left(n/2\right) + \mathcal{O}\left(1\right) & \text{if } n > 1. \end{array}$$

where, γ is a constant. The cache complexity of a subproblem of size n when it fits cache i.e., $n \le \gamma M$, is $\Theta\left(n/B+1\right) = \mathcal{O}\left(M/B+1\right)$. The cache complexity of a subproblem when it does not fit into cache is recursively computed using its subproblems. The cache complexity recurrence can be solved using an approach similar to the one given in the proof of Theorem 4.2 and by replacing $\log 3$ with $\log 4$. The span recurrence can be solved by using the master theorem [33, 32] first on the Partition function and then on BubbleSort. □

SelectionSort-Iterative($A[0..n-1]$)

```
for i ← 0 to n − 2 do
   min ← i
   for j ← i + 1 to n − 1 do
      if A[j] < A[min] then
         min ← j
   Swap(A[i], A[min])
```

SelectionSort($A[\ell..h]$)

Sorts a given array by parallel divide-and-conquer selection sort
Input: Subarray $A[\ell..h]$ of orderable elements
Output: $A[\ell..h]$ sorted in nondecreasing order

```
if (h − ℓ + 1) ≤ b then
   for i ← ℓ to h − 1 do
      min ← i
      for j ← i + 1 to h do
         if A[j] < A[min] then
            min ← j
      if min ≠ i then
         Swap(A[i], A[min])
else
   m ← (ℓ + h)/2
   Partition(A[ℓ..m], A[m + 1..h])
   par: SelectionSort(A[ℓ..m]),
        SelectionSort(A[m + 1..h])
```

Partition($A[\ell\ell..\ell h], A[r\ell..rh]$)

Partitions elements between two subarrays without using a pivot
Input: Two equal sized non-overlapping subarrays $A[\ell\ell..\ell h]$ and $A[r\ell..rh]$
Output: Partition the elements such that all elements in $A[\ell\ell..\ell h]$ are less than or equal to all elements in $A[r\ell..rh]$

```
if (ℓh − ℓℓ + 1) ≤ b then
   for i ← ℓℓ to ℓh do
      min ← i
      for j ← rℓ to rh do
         if A[j] < A[min] then
            min ← j
      if min ≠ i then
         Swap(A[i], A[min])
else
   ℓm ← (ℓℓ + ℓh)/2; rm ← (rℓ + rh)/2
   par: Partition(A[ℓℓ..ℓm], A[rℓ..rm]),
        Partition(A[ℓm+1..ℓh], A[rm+1..rh])
   par: Partition(A[ℓℓ..ℓm], A[rm + 1..rh]),
        Partition(A[ℓm + 1..ℓh], A[rℓ..rm])
```

FIGURE 2. A recursive divide-and-conquer selection sort algorithm. Initial call to the recursive algorithm is SelectionSort($A[0..n-1]$), where $A[0..n-1]$ is the array to be sorted.

3. SELECTION SORT

Selection sort is another slow running sorting algorithm that sorts n numbers in $\mathcal{O}\left(n^2\right)$ time.

An iterative algorithm SELECTIONSORT-ITERATIVE is given in Figure 2. The algorithm has n iterations. In each iteration $i \in [0, n-1]$, the position of the minimum element, denoted by min is found in the range $A[i..n-1]$. Then the elements $A[min]$ and $A[i]$ are swapped. The algorithm makes sure that after iteration i, the ith smallest element is in its correct position.

A recursive divide-and-conquer selection sort algorithm SELECTIONSORT is shown in Figure 2. The initial invocation to the algorithm is SELECTIONSORT($A[0..n-1]$). The recursive structure of the algorithm is exactly the same as that of bubble sort. The SELECTIONSORT function invokes the PARTITION function to partition the array A into two halves where the largest element in the first half is lesser than or equal to the smallest element in the second half. After the partition, the SELECTIONSORT functions are invoked on the two halves to sort them recursively. The partition function PARTITION calls itself four times in two parallel steps. The only difference between the bubble sort and selection sort divide-and-conquer algorithms are the base cases of SELECTIONSORT and PARTITION functions.

The base case kernel of SELECTIONSORT function is equivalent to SELECTIONSORT-ITERATIVE. In the base case kernel of the PARTITION function, in each iteration, an element that is lesser than or equal to b elements is pushed to the left subarray. After several iterations, the elements in the two subarrays would be partitioned in such a way that the largest element in the left subarray $A[\ell\ell..\ell h]$ would be lesser than or equal to the smallest element in the right subarray $A[r\ell..rh]$.

The proof of correctness and complexity analysis of SELECTIONSORT are given in Theorems 3.1 and 3.2, respectively.

THEOREM 3.1 (**Selection Sort Correctness**). SELECTIONSORT *correctly sorts an unsorted array.*

Proof. We use mathematical induction to prove the theorem. First we prove the correctness of PARTITION function. Then we prove SELECTIONSORT correct. We assume that n and b are powers of 2 such that $n \geq b$. We say $A[\ell\ell..\ell h]$ and $A[r\ell..rh]$ as left and right input subarrays, respectively.

(1) [Correctness of PARTITION.]
Basis. The logic of the base case when the input subarray is of size b is straightforward. The external loop runs b times. In each iteration, we find the index of the smallest element in the right subarray and if that element is less than an element in the left subarray, then we swap the two elements. In this way, the smallest b elements will move to the left subarray.
Induction. The argument is similar to that given in Theorem 2.2.

(2) [Correctness of SELECTIONSORT.]
Basis. The base case when the subarray is of size b is exactly same as the standard iterative selection sort.
Induction. The argument is similar to the one in Theorem 2.2. □

THEOREM 3.2 (**Selection Sort Complexity**). SELECTIONSORT *incurs $\mathcal{O}(n^2/(BM) + n/B + 1)$ cache misses and has $\Theta(n)$ span.*

Proof. Let $Q_1^f(n)$ and $T_\infty^f(n)$ denote the number of serial cache misses and span of algorithm f, respectively. Let SS-I, SS, and PART denote SELECTIONSORT-ITERATIVE, SELECTIONSORT, and PARTITION, respectively. Then,

$$Q_1^{\text{SS-I}}(n) = \textstyle\sum_{i=0}^{n-1} \Theta\left(((n-i)/B) + 1\right) = \Theta\left(n^2/B + n\right).$$

$$\begin{array}{ll} Q_1^{\text{SS}}(n) = Q_1^{\text{PART}}(n) = \mathcal{O}\left(n/B + 1\right) & \text{if } n \leq \gamma M, \\ Q_1^{\text{SS}}(n) = 2Q_1^{\text{SS}}\left(n/2\right) + Q_1^{\text{PART}}\left(n/2\right) + \mathcal{O}\left(1\right) & \text{if } n > \gamma M, \\ Q_1^{\text{PART}}(n) = 4Q_1^{\text{PART}}\left(n/2\right) + \mathcal{O}\left(1\right) & \text{if } n > \gamma M. \end{array}$$

$$\begin{array}{ll} T_\infty^{\text{SS}}(n) = T_\infty^{\text{PART}}(n) = \mathcal{O}\left(1\right) & \text{if } n = 1, \\ T_\infty^{\text{SS}}(n) = T_\infty^{\text{SS}}\left(n/2\right) + T_\infty^{\text{PART}}\left(n/2\right) + \mathcal{O}\left(1\right) & \text{if } n > 1, \\ T_\infty^{\text{PART}}(n) = 2T_\infty^{\text{PART}}\left(n/2\right) + \mathcal{O}\left(1\right) & \text{if } n > 1. \end{array}$$

where, γ is a suitable constant. The cache complexity recurrence can be solved using an approach similar to the one given in the proof of Theorem 4.2 and by replacing $\log 3$ with $\log 4$. The span recurrence can be solved by using the master theorem [33, 32] first on the PARTITION function and then on SELECTIONSORT. □

4. INSERTION SORT

Insertion sort is a pretty fast algorithm compared with other elementary sorting algorithms, which sorts a set of n elements in $\mathcal{O}\left(n^2\right)$ worst and average case time.

An iterative algorithm INSERTIONSORT-ITERATIVE is given in Figure 3. The algorithm has $n-1$ iterations. In iteration $i \in [1, n-1)$, the array element $A[i]$ will be inserted in a sorted position in the range $A[0..i-1]$. After each iteration i, the subarray $A[0..i]$ will be sorted. The runtime complexity is data-sensitive.

A recursive divide-and-conquer insertion sort algorithm INSERTIONSORT is shown in Figure 3. The initial invocation to the algorithm is INSERTIONSORT($A[0..n-1]$). The recursive structure of the algorithm is different from that of bubble and selection sorts. The INSERTIONSORT function calls itself twice to sort the left and right halves separately and simultaneously. Then it invokes the MERGE function to merge the elements from the two halves using the logic of the iterative insertion sort. After the merge, the entire array would be sorted.

The merge function MERGE calls itself a total of three times: the first two calls in parallel and then a third serial call. The first call MERGE($A[\ell\ell..\ell m], A[r\ell..rm]$) brings the smallest elements to $A[\ell\ell..\ell m]$ in sorted order. The second call MERGE($A[\ell m+1..\ell h], A[rm+1..rh]$) brings the largest elements to $A[rm+1..rh]$ in sorted order. The third call MERGE($A[\ell m+1..\ell h], A[r\ell..rm]$) brings the remaining elements to $A[\ell m+1..rm]$ in the sorted order.

InsertionSort-Iterative($A[0..n-1]$)

```
for i ← 1 to n − 1 do
  key ← A[i]
  j ← i − 1
  while j ≥ 0 and A[j] > key do
    A[j + 1] ← A[j]
    j ← j − 1
  A[j + 1] ← key
```

InsertionSort($A[\ell..h]$)

Sorts a given array by parallel divide-and-conquer insertion sort
Input: Subarray $A[\ell..h]$ of orderable elements
Output: $A[\ell..h]$ sorted in nondecreasing order

```
if (h − ℓ + 1) ≤ b then
  for i ← ℓ + 1 to h do
    key ← A[i]
    j ← i − 1
    while j ≥ ℓ and A[j] > key do
      A[j + 1] ← A[j]
      j ← j − 1
    A[j + 1] ← key
else
  m ← (ℓ + h)/2
  par: InsertionSort(A[ℓ..m]),
       InsertionSort(A[m + 1..h])
  Merge(A[ℓ..m], A[m + 1..h])
```

Merge($A[\ell\ell..\ell h], A[r\ell..rh]$)

Merges two sorted arrays into a sorted list
Input: Two equal sized non-overlapping subarrays $A[\ell\ell..\ell h]$ and $A[r\ell..rh]$ both sorted
Output: Sorted list of the elements from the two input subarrays $A[\ell\ell..\ell h]$ and $A[r\ell..rh]$; Merge is performed in-place

```
1.  if (ℓh − ℓℓ + 1) ≤ b then
2.    if A[ℓh] > A[rℓ] then
3.      for i ← rℓ to rh do
4.        key ← A[i]; j ← i − 1
5.        while j ≥ rℓ and A[j] > key do
6.          A[j + 1] ← A[j]; j ← j − 1
7.        if A[ℓh] > key then
8.          A[rℓ] ← A[ℓh]; j ← lh − 1
9.          while j ≥ ℓℓ and A[j] > key do
10.           A[j + 1] ← A[j]; j ← j − 1
11.       A[j + 1] ← key
12. else
13.   ℓm ← (ℓℓ + ℓh)/2; rm ← (rℓ + rh)/2
14.   par: Merge(A[ℓℓ..ℓm], A[rℓ..rm]),
           Merge(A[ℓm + 1..ℓh], A[rm + 1..rh])
15.   Merge(A[ℓm + 1..ℓh], A[rℓ..rm])
```

FIGURE 3. A recursive divide-and-conquer insertion sort algorithm. Initial call to the recursive algorithm is InsertionSort($A[0..n-1]$), where $A[0..n-1]$ is the array to be sorted.

The base case kernel of InsertionSort function is equivalent to InsertionSort-Iterative. The base case kernel of the Merge function merges two sorted subarrays to a sorted array. In iteration k, the kth element of the right subarray gets merged with its previous elements in the right subarray and with the elements of the left subarray. After $rh - r\ell + 1$ iterations, the elements in the two subarrays would be merged into a sorted array – the left subarray will be sorted, the right subarray will be sorted, and the last element of the left subarray will be less than or equal to the first element of the right subarray.

The proof of correctness and complexity analysis of InsertionSort are given in Theorems 4.1 and 4.2, respectively.

Theorem 4.1 (**Insertion Sort Correctness**). InsertionSort *correctly sorts an unsorted array.*

Proof. We use mathematical induction to prove the theorem. First we prove the correctness of the Merge function. Then we prove InsertionSort correct. We assume that n and b are powers of 2 such that $n \geq b$. We say $A[\ell\ell..\ell h]$ and $A[r\ell..rh]$ as left and right input subarrays, respectively.

(1) [Correctness of Merge.]
Basis. The logic of the base case when the input subarray is of size b is straightforward. The external loop runs for all elements in the right subarray i.e., b times. In each iteration, the element from the right subarray is inserted into its correct position towards its left by shifting elements.
Induction. We assume that Merge works correctly when the input subarrays are of size 2^k for some k, such that $2^k \geq b$. We need to prove that Merge works for input subarrays of size 2^{k+1}.

Let Q_1, Q_2, Q_3, and Q_4, where Q stands for "quarter", represent the subarrays $A[\ell\ell..\ell m], A[(\ell m + 1)..\ell h], A[r\ell..rm]$, and $A[(rm+1)..rh]$, respectively, where each subarray is of size 2^k. Let W, X, Y, and Z be the initial sets (not lists) of numbers present at Q_1, Q_2, Q_3, and Q_4, respectively. Let Small(S_1, S_2) (resp. Large(S_1, S_2)) of two equal-sized sets S_1 and S_2 of numbers represent a set consisting of the smallest half (resp. largest half) of the numbers from sets S_1 and S_2. Also, let $S_1 \leq S_2$ denote that all elements of S_1 is less than or equal to all elements of S_2. As the input subarrays are sorted we have $W \leq X$. Hence, we can write $W =$ Small(W, X) and $X =$ Large(W, X). Similarly, $Y \leq Z$. Therefore, we can write $Y =$ Small(Y, Z) and $Z =$ Large(Y, Z).

Consider the Merge function in Figure 3. After execution of line 13, the states of the four quarters of the array A are $Q_1 = W$, $Q_2 = X$, $Q_3 = Y$, and $Q_4 = Z$. After execution of line 14, the states of the four quarters of the array A are: $Q_1 =$ Small(W, Y), $Q_2 =$ Small(X, Z), $Q_3 =$ Large(W, Y), and $Q_4 =$ Large(X, Z). After execution of line 15, the states of the four quarters of the array A are:

$$Q_1 = \textsc{Small}(W, Y)$$
$$Q_2 = \textsc{Small}(\textsc{Small}(X, Z), \textsc{Large}(W, Y))$$
$$Q_3 = \textsc{Large}(\textsc{Small}(X, Z), \textsc{Large}(W, Y))$$
$$Q_4 = \textsc{Large}(X, Z)$$

It is easy to see that

$$\begin{aligned} Q_1 &= \textsc{Small}(\textsc{Small}(W, X), \textsc{Small}(Y, Z)) \\ &= \textsc{Small}(\textsc{Small}(W, Y), \textsc{Small}(X, Z)) \leq Q_2 \\ Q_2 &\leq Q_3 \\ Q_3 &\leq \textsc{Large}(\textsc{Large}(X, Z), \textsc{Large}(W, Y)) \\ &= \textsc{Large}(\textsc{Large}(W, X), \textsc{Large}(Y, Z)) = Q_4 \end{aligned}$$

As $Q_1 \leq Q_2 \leq Q_3 \leq Q_4$, the input subarrays of size 2^{k+1} have been merged.

(2) [Correctness of InsertionSort.]
Basis. The base case when the input subarray is of size b is exactly same as the standard iterative insertion sort.
Induction. We assume that InsertionSort works correctly when the input subarrays are of size 2^k for some k, such that $2^k \geq b$. We need to prove that InsertionSort works for input subarrays of size 2^{k+1}. We know that the InsertionSort function is correct. Hence, after line 11, the left subarray ($A[\ell..m]$) would be sorted and the right subarray ($A[(m+1)..h]$) would be sorted. Then after line 12, we merge the two subarrays. As we have shown that the merge function Merge is correct, the entire subarray of size 2^{k+1} would be merged and sorted.

□

Theorem 4.2 (**Insertion Sort Complexity**). InsertionSort *performs* $\mathcal{O}\left(n^{\log_2 3}\right)$ *work, incurs* $\mathcal{O}(n^{\log_2 3}/(BM^{(\log_2 3)-1}) + n/B + 1)$ *cache misses, and has* $\mathcal{O}(n)$ *span.*

Proof. Let $T_1^f(n)$, $Q_1^f(n)$, and $T_\infty^f(n)$ denote the work, the number of serial cache misses, and span of algorithm f, respectively. Let IS-I, IS, and Merge denote InsertionSort-Iterative, InsertionSort, and Merge, respectively. Then,

$$Q_1^{\text{IS-I}}(n) = \textstyle\sum_{i=0}^{n-1} \mathcal{O}\left(((n-i)/B) + 1\right) = \mathcal{O}\left(n^2/B + n\right).$$

$$\begin{aligned} &T_1^{\text{IS}}(n) = T_1^{\text{Merge}}(n) = \mathcal{O}(1) && \text{if } n = 1, \\ &T_1^{\text{IS}}(n) = 2T_1^{\text{IS}}(n/2) + T_1^{\text{Merge}}(n/2) + \Theta(1) && \text{if } n > 1, \\ &T_1^{\text{Merge}}(n) = 3T_1^{\text{Merge}}(n/2) + \Theta(1) && \text{if } n > 1. \end{aligned}$$

$$\begin{aligned} &Q_1^{\text{IS}}(n) = Q_1^{\text{Merge}}(n) = \mathcal{O}(n/B + 1) && \text{if } n \leq \gamma M, \\ &Q_1^{\text{IS}}(n) = 2Q_1^{\text{IS}}(n/2) + Q_1^{\text{Merge}}(n/2) + \mathcal{O}(1) && \text{if } n > \gamma M, \\ &Q_1^{\text{Merge}}(n) = 3Q_1^{\text{Merge}}(n/2) + \mathcal{O}(1) && \text{if } n > \gamma M. \end{aligned}$$

$$\begin{aligned} &T_\infty^{\text{IS}}(n) = T_\infty^{\text{Merge}}(n) = \mathcal{O}(1) && \text{if } n = 1, \\ &T_\infty^{\text{IS}}(n) = T_\infty^{\text{IS}}(n/2) + T_\infty^{\text{Merge}}(n/2) + \mathcal{O}(1) && \text{if } n > 1, \\ &T_\infty^{\text{Merge}}(n) = 2T_\infty^{\text{Merge}}(n/2) + \mathcal{O}(1) && \text{if } n > 1. \end{aligned}$$

where, γ is a suitable constant. The span recurrence can be solved by using the master theorem [33, 32] first on the Merge function and then on InsertionSort.

The derivation of the serial cache complexity of IS is given as follows. We initially find Q_1^{Merge} and then use it to compute Q_1^{IS}. We assume that $n/2^k = \gamma M$ for some γ and all logarithms are taken to the base 2.

$$\begin{aligned} Q_1^{\text{Merge}}(n) &= 3Q_1^{\text{Merge}}\left(\frac{n}{2}\right) + \Theta(1) \\ &= 3^k Q_1^{\text{Merge}}\left(\frac{n}{2^k}\right) + \Theta(1)\left(3^{k-1} + \cdots + 3^0\right) \\ &= 3^k\left(Q_1^{\text{Merge}}\left(\frac{n}{2^k}\right) + \Theta(1)\right) \\ &= \mathcal{O}\left(\left(\frac{n}{M}\right)^{\log 3}\left(\frac{M}{B}\right)\right) \\ &= \mathcal{O}\left(\frac{n^{\log 3}}{BM^{\log 3 - 1}}\right) \end{aligned}$$

Plugging Q_1^{Merge} into the recurrence of Q_1^{IS}, we get

$$\begin{aligned} Q_1^{\text{IS}}(n) &= 2Q_1^{\text{IS}}\left(\frac{n}{2}\right) + Q_1^{\text{Merge}}\left(\frac{n}{2}\right) + \Theta(1) \\ &= 2^k\left(Q_1^{\text{IS}}\left(\frac{n}{2^k}\right) + \Theta(1)\right) \\ &\quad + \Theta(1)\cdot\left(\frac{n}{M}\right)^{\log 3}\left(\frac{M}{B}\right)\cdot\sum_{i=1}^{k}\left(\frac{1}{2^{(i-1)(\log 3 - 1)}}\right) \\ &= \mathcal{O}\left(\frac{n}{M}\left(\frac{M}{B}\right) + \left(\frac{n}{M}\right)^{\log 3}\left(\frac{M}{B}\right)\right) \\ &= \mathcal{O}\left(\frac{n^{\log 3}}{BM^{\log 3 - 1}} + \frac{n}{B} + 1\right) \end{aligned}$$

Solving the recurrences we get the theorem. □

5. EXPERIMENTS

This section presents empirical results showing performance improvements of the divide-and-conquer sorting algorithms from high parallelism and better cache complexity over their iterative counterparts.

Setup. Our experiments were performed on a multicore machine with dual-socket 8-core 2.7 GHz Intel Sandy Bridge processors ($2 \times 8 = 16$ cores in total) and 32 GB RAM. Each core was linked to a 32 KB private L1 cache and a 256 KB private L2 cache. All cores in a processor shared a 20 MB L3 cache. With hyper-threading, we can simulate a total of 32 threads from 16 cores. The algorithms were implemented in C++. Intel Cilk Plus extension was used to parallelize the programs. Intel C++ Compiler v13.0 (`icc`) was used to compile the implementations with parameters `-O3 -ipo -parallel -AVX -xhost`. Apart from these parameters no optimizations were used.

Implementations. The three standard iterative sorting algorithms: BubbleSort-Iterative, SelectionSort-Iterative, and InsertionSort-Iterative were implemented without any optimization and the implementations were inherently serial.

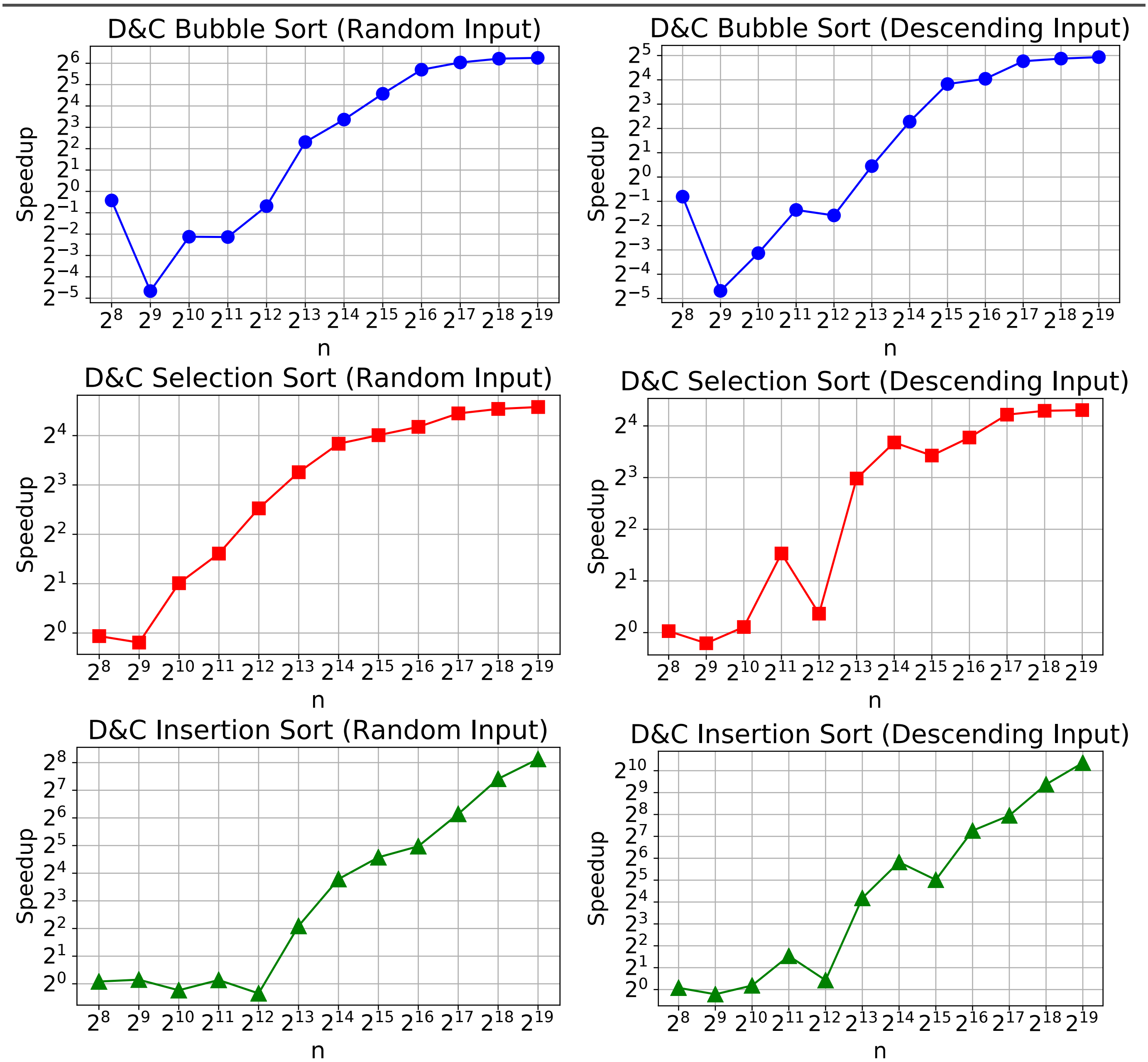

FIGURE 4. Speedup of the divide-and-conquer bubble, selection, and insertion sorts for (i) random input (left column) and (ii) descending order input (right column). The definition of speedup is given in Equation 1.

The divide-and-conquer algorithms: BUBBLESORT, SELECTIONSORT, and INSERTIONSORT were also implemented without optimizations. When the subproblem size ($h-\ell+1$ or $\ell h-\ell\ell+1$) became less than or equal to a base case size $b = 2^8 = 256$, we switched to an iterative kernel having an algorithm-dependent logic. The recursive algorithms were run with 32 threads. We define speedup as follows:

$$\text{Speedup} = \frac{\text{Runtime of iterative algorithm}}{\text{Runtime of recursive algo. with 32 threads}} \tag{1}$$

In our experiments, we used two types of input: (i) random input (using `rand` function), and (ii) descending order input. The input size n was varied from $2^8 = 256$ to $2^{19} = 524288$ and the speedup was computed for different algorithms based on Equation 1.

Results. Table 4 shows the speedup graphs for the three sorting algorithms.

The speedup of the BUBBLESORT program increased from 0.7× to 76× for the random input when n increased. The super-linear speedup is due to cache-optimality. For the descending order input, the speedup increased from 0.6× to 30.6×. The good speedup is majorly due to parallelism and to some extent by the cache performance.

The SELECTIONSORT program speedup increased from $\approx 1\times$ to $23.9\times$ for the random input when n increased. When the input was in decreasing order, the speedup increased from $\approx 1\times$ to $19.8\times$. Compared to bubble sort, the speedup is less for selection sort. The reason is that SELECTIONSORT does more number of comparisons than SELECTIONSORT-ITERATIVE.

For the random input, the speedup of the INSERTIONSORT program increased from $1.1\times$ to $280\times$. For the decreasing order input, the speedup increased from $1.1\times$ to $1314.7\times$. Note that such a large speedup is not possible from parallelism and cache performance alone. The factor that is increasing the speedup is the asymptotic less work that the INSERTIONSORT performs compared to INSERTIONSORT-ITERATIVE as shown in Theorem 4.2.

6. CONCLUSION

We presented parallel divide-and-conquer algorithms for bubble sort, selection sort, and insertion sort. The algorithms are fast (i.e., cache-efficient and parallel) and portable (i.e., cache-oblivious). The implementations of our algorithms run significantly faster than their iterative counterparts. It would be interesting to know if we can design parallel divide-and-conquer algorithms related to more sorting techniques.

7. FUNDING

This work was supported in part by National Science Foundation grants [CCF-1162196, CCF-1439084, CNS-1553510].

8. DATA AVAILABILITY

No new input data were generated or analysed in support of this research.

REFERENCES

[1] Knuth, D. E. (1998) *The art of computer programming: sorting and searching.* Pearson Education.

[2] Skiena, S. S. (1998) *The algorithm design manual.* Springer Science and Business Media.

[3] Akl, S. G. (2014) *Parallel sorting algorithms.* Academic press.

[4] Cole, R. (1988) Parallel merge sort. *SIAM Journal on Computing*, **17(4)**, 770–785.

[5] Estivill-Castro, V. and Wood, D. (1992) A survey of adaptive sorting algorithms. *ACM Computing Surveys*, **24(4)**, 441–476.

[6] Ailon, N., Chazelle, B., Clarkson, K. L., Liu, D., Mulzer, W., and Seshadhri, C. (2011) Self-improving algorithms. *SIAM Journal on Computing*, **40(2)**, 350–375.

[7] Levitin, A. (2011) *Introduction to the Design and Analysis of Algorithms*, third edition. Pearson.

[8] Chatterjee, S., Lebeck, A. R., Patnala, P. K., and Thottethodi, M. (2002) Recursive array layouts and fast matrix multiplication. *IEEE Transactions on Parallel and Distributed Systems*, **13(11)**, 1105–1123.

[9] Frens, J. D. and Wise, D. S. (1997) Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. *ACM Symposium on Principles and Practice of Parallel Programming*, Las Vegas, USA, 18-21 June, pp. 32(7):206–216. ACM.

[10] Chowdhury, R., Ganapathi, P., Tschudi, S., Tithi, J. J., Bachmeier, C., Leiserson, C. E., Solar-Lezama, A., Kuszmaul, B. C., and Tang, Y. (2017) Autogen: Automatic Discovery of Efficient Recursive Divide-&-Conquer Algorithms for Solving Dynamic Programming Problems. *ACM Transactions on Parallel Computing*, **4**, 1–30.

[11] Chowdhury, R., Ganapathi, P., Pradhan, V., Tithi, J. J., and Xiao, Y. (2016) An Efficient Cache-Oblivious Parallel Viterbi Algorithm. *Proceedings of the 22nd European Conference on Parallel Processing*, Grenoble, France, 22-26 August, pp. 574–587. Springer.

[12] Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. (2012) Cache-oblivious algorithms. *ACM Transactions on Algorithms*, **8**, 1–22.

[13] Bender, M. A., Ebrahimi, R., Fineman, J. T., Ghasemiesfeh, G., Johnson, R., and McCauley, S. (2014) Cache-adaptive algorithms. *Proceedings of the 25th ACM-SIAM Symposium on Discrete Algorithms*, Portland, USA, 05-07 January, pp. 958–971. SIAM ACM.

[14] Bender, M. A., Demaine, E. D., Ebrahimi, R., Fineman, J. T., Johnson, R., Lincoln, A., Lynch, J., and McCauley, S. (2016) Cache-adaptive analysis. *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, Pacific Grove, USA, 11-13 July, pp. 135–144. ACM.

[15] Bentley, J. L. (1980) Multidimensional divide-and-conquer. *Communications of the ACM*, **23(4)**, 214–229.

[16] Mou, Z. G. and Hudak, P. (1988) An algebraic model for divide-and-conquer and its parallelism. *The Journal of Supercomputing*, **2(3)**, 257–278.

[17] Chowdhury, R., Ganapathi, P., Tang, Y., and Tithi, J. J. (2017) Provably Efficient Scheduling of Cache-Oblivious Wavefront Algorithms. *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, Washington DC, USA, 24-26 July, pp. 339–350. ACM.

[18] Ganapathi, P. (2016) Automatic Discovery of Efficient Divide-&-Conquer Algorithms for Dynamic Programming Problems. PhD thesis Department of Computer Science, State University of New York at Stony Brook.

[19] Ahmad, Z., Chowdhury, R., Das, R., Ganapathi, P., Gregory, A., and Zhu, Y. (2021) Fast stencil computations using fast Fourier transforms. *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, Virtual Event, USA, 6-8 July. ACM.

[20] Javanmard, M. M., Ganapathi, P., Das, R., Ahmad, Z., Tschudi, S., and Chowdhury, R. (2019) Toward Efficient Architecture-Independent Algorithms for Dynamic Programs. *International Conference on High Performance Computing*, Frankfurt, Germany, 16-20 June, pp. 143–164. Springer.

[21] JáJá, J. (1992) *An Introduction to Parallel Algorithms.* Addison-Wesley.

[22] Friend, E. H. (1956) Sorting on electronic computer systems. *Journal of the ACM*, **3(3)**, 134–168.

[23] Gotlieb, C. (1963) Sorting on computers. *Communications of the ACM*, **6(5)**, 194–201.

[24] Habermann, N. (1972) Parallel neighbor-sort (or the glory of the induction principle). *CMU Technical Report*, **1**, 1–12.

[25] (2008). Paul E. Black. “Bingo Sort”. Dictionary of Algorithms and Data Structures. `http://xlinux.nist.gov/dads//HTML/bingosort.html`. [Online; accessed 26-February-2020].

[26] Shell, D. L. (1959) A high-speed sorting procedure. *Communications of the ACM*, **2(7)**, 30–32.

[27] Williams, J. W. J. (1964) Algorithm 232: Heapsort. *Commun. ACM*, **7**, 347–348.

[28] Bender, M. A., Farach-Colton, M., and Mosteiro, M. A. (2006) Insertion sort is $O(n \log n)$. *Theory of Computing Systems*, **39(3)**, 391–397.

[29] LaMarca, A. and Ladner, R. E. (1999) The influence of caches on the performance of sorting. *Journal of Algorithms*, **31(1)**, 66–104.

[30] Aggarwal, A. and Vitter, J. (1988) The input/output complexity of sorting and related problems. *Communications of the ACM*, **31(9)**, 1116–1127.

[31] Blelloch, G. E., Fineman, J. T., Gu, Y., and Sun, Y. (2020) Optimal parallel algorithms in the binary-forking model. *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, Virtual Event, USA, 15-17 July, pp. 89–102. ACM.

[32] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009) *Introduction to algorithms.* MIT press.

[33] Bentley, J. L., Haken, D., and Saxe, J. B. (1980) A general method for solving divide-and-conquer recurrences. *ACM SIGACT News*, **12**, 36–44.