# Sorting by Parallel Insertion on a One-Dimensional Sub-Bus Array

James D. Fix and Richard E. Ladner

Technical Report # 96-09-02

Department of Computer Science and Engineering
University of Washington, Seattle 98195

## Abstract

We consider the problem of sorting on a one-dimensional sub-bus array of processors. The sub-bus broadcast operation makes possible a new class of parallel sorting algorithms whose complexity we analyze with the *parallel insertion model.* A sorting method, or *sorting strategy*, in the parallel insertion model, uses a sequence of *left and right insertion steps*, of which we give two types: *greedy insertion steps* and *simple insertion steps.* For two restricted classes of parallel insertion sorting, the *one-way* and the *alternating* sorting strategies, we give lower bounds and optimal sorting strategies that exactly match the lower bounds. Optimal alternating sorting strategies are demonstrated to use a factor of two fewer insertion steps on average than odd-even transposition sort and any optimal one-way sorting strategy. For general sorting strategies, we give a weak lower bound and consider a sorting strategy that uses the fewest greedy insertion steps. Finally, we discuss the issues involved in implementing parallel insertion sorting strategies on sub-bus machines. We evaluate the performance of our sorting strategies by applying them to shearsort, a common two-dimensional mesh sorting algorithm, and by contrasting the results with our theoretical results from the parallel insertion model.

# Sorting by Parallel Insertion on a One-Dimensional Sub-Bus Array[*]

James D. Fix[†]
Richard E. Ladner[‡]

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

September 16, 1996

**Abstract**

We consider the problem of sorting on a one-dimensional sub-bus array of processors. The sub-bus broadcast operation makes possible a new class of parallel sorting algorithms whose complexity we analyze with the *parallel insertion model*. A sorting method, or *sorting strategy*, in the parallel insertion model, uses a sequence of *left and right insertion steps*, of which we give two types: *greedy insertion steps* and *simple insertion steps*. For two restricted classes of parallel insertion sorting, the *one-way* and the *alternating* sorting strategies, we give lower bounds and optimal sorting strategies that exactly match the lower bounds. Optimal alternating sorting strategies are demonstrated to use a factor of two fewer insertion steps on average than odd-even transposition sort and any optimal one-way sorting strategy. For general sorting strategies, we give a weak lower bound and consider a sorting strategy that uses the fewest greedy insertion steps. Finally, we discuss the issues involved in implementing parallel insertion sorting strategies on sub-bus machines. We evaluate the performance of our sorting strategies by applying them to shearsort, a common two-dimensional mesh sorting algorithm, and by contrasting the results with our theoretical results from the parallel insertion model.

## 1 Introduction

A one-dimensional sub-bus array has a bus connecting the processors which can be segmented into sub-buses on which an active processor can broadcast. The sub-bus broadcast capability has

[*]A preliminary version of this work was presented in the Sixth Annual Symposium on Discrete Algorithms, 1995 [6]. Use of the MasPar MP-1 for this research was made possible by the Parallel Processing Laboratory at the Purdue University School of Electrical and Computer Engineering and NSF award number CDA-9015696.

[†]fix@cs.washington.edu. Fix's research was supported by an Osberg Family Trust Fellowship and by the National Science Foundation under Grant CCR-9301186.

[‡]ladner@cs.washington.edu.

been implemented on the MasPar MP-1 and MP-2 parallel computers [1]. The MasPar computers are two-dimensional arrays of processors which are controlled by a SIMD parallel program. One-dimensional sub-bus operations are available in each of eight directions with wrap around. Many parallel algorithms have been developed for the processor array model where only nearest neighbor communication is allowed, and these algorithms are applicable to the sub-bus array model as well. However, a natural question to ask is whether the sub-bus capability gives rise to better algorithms than are possible on the standard mesh of processors.

The purpose of this paper is to try to determine the best way to sort in place and along one dimension using the sub-bus broadcast operations. We present models and techniques for developing sub-bus sorting algorithms and for showing the limitations of sorting on a sub-bus array. For several categories of sub-bus sorting methods we give exact optimality criteria and present sorting strategies that match those criteria. One advantage of our techniques is that we are able to provide average case analysis for our sub-bus sorting strategies. For some sorting strategies which defy analysis we use simulation to compare their performances. Finally, we implement several sorting strategies to see how well they perform in a real sorting environment. We compare our sorting methods with odd-even transposition sort, a well-known sort used on standard processor arrays.

## 1.1 Summary of Results

We begin by describing the parallel insertion model that models one-dimensional, in-place, comparison based, sorting on a sub-bus array (section 2). A left (or right) insertion step in the parallel insertion model resembles a set of simultaneous data insertions. A sorting strategy (a generalization of a sorting algorithm) is a sequence of insertion steps that sorts a given input. We describe two types of left and right insertion steps, greedy insertion steps and simple insertion steps, of which all our strategies are composed. The greedy insertion steps have the interesting property that they remove the maximum number of inversions possible by any insertion step. Greedy insertion steps are not efficiently implementable, requiring $\log_2 n$ sub-bus operations per step, while simple insertion steps are efficiently implementable, using just three sub-bus operations per step.

We first describe one-way sorting strategies whose insertion steps are always in the same direction. We show that that the number of steps required to sort using a sorting strategy that uses only left insertion steps is at least the maximum distance a data value is from its final destination for those values which are left of their final destinations. We show that if all permutations of the data are equally likely then the expected number of steps required to sort with a one-way sorting strategy is $n - \sqrt{\frac{\pi n}{2}} + \frac{2}{3} - \Theta(n^{-\frac{1}{2}})$ where $n$ is the number of processors in the array. We show that the one-way sorting strategies consisting of greedy insertion steps and of simple insertion steps use exactly the required number of steps. These exact bounds on the expected number of steps for one-way sorting strategies indicates that the advantage of these sorting strategies over odd-even transposition sort is quite small on average.

We next examine two-way sorting strategies which allow both left and right insertion steps (section 4). The fundamental limitation of sub-bus architectures is that when the bus is segmented into sub-buses only one data value can be communicated on each sub-bus within each time step. A similar limitation exists for insertion steps as well. In section 4.1 we define a metric on permutations which gives the maximum number of data values which must traverse a cut in the array in order to reach their final destinations. We show that the expected number of insertion steps required to to sort by a two-way sorting strategy is at least $n/4$. A special case of two-way sorting strategies are alternating sorting strategies in which left insertion steps alternate with right insertion steps. We show that the number of steps required to sort using an alternating sorting strategy is directly related to our metric. We extend our bound for general strategies to show that the expected number of insertion steps required to sort by an alternating strategy is at least $n/2$. We prove that an alternating sorting strategy that uses greedy insertion steps is optimal but one that uses simple insertion steps is not optimal.

In section 6 we use simulation to investigate the speed of alternating simple insertion sort. Our simulations indicate that alternating simple insertion sort is almost as fast as the optimal alternating greedy sort and both are almost twice as fast as the standard odd-even transposition sort. Finally, we consider the best sorting strategy we can think of, namely, a best-greedy sort, which sorts in the minimum number of greedy insertion steps. A best-greedy sort is only slightly faster than our alternating sorting strategies. Our simulations indicate that average performance of alternating greedy, alternating simple, and best-greedy sorts are almost identical for large $n$, taking slightly more than $n/2$ steps on average. Figure 7 summarizes our simulation results.

Finally, we describe how to implement our sorting strategies on a real sub-bus machine (section 7). Interestingly, a left (or right) simple insertion step can be expressed almost as simply as a step in odd-even transposition sort. In section 8 we give the results of implementing alternating simple insertion sort as the one-dimensional sort in the two-dimensional shearsort algorithm. We compare alternating simple insertion sort with two versions of odd-even transposition sort, an early-stopping version that stops when the array is sorted and an oblivious version. Surprisingly, shearsort using early-stopping odd-even transposition sort uses nearly as few insertion steps as shearsort using alternating simple insertion sort, which is nearly half that of shearsort using the oblivious odd-even transposition sort. On the MasPar MP-1, oblivious odd-even transposition steps are nearly twice as fast as simple insertion steps with early-stopping odd-even transposition steps falling in between the two. The end result is that the versions of shearsort that use oblivious odd-even transposition steps and simple insertion steps perform similarly but shearsort using early-stopping odd-even transposition sort performs best.

## 1.2 Related Work

Odd-even transposition sort [7, 10] is used for sorting on linear arrays without sub-buses where only nearest neighbor communication is possible. In odd-even transposition sort, processors swap values with their adjacent processors until order is achieved. It is commonly used as a subcomponent of optimal sorting algorithms for a two-dimensional array of processors [15, 16, 17]. In a similar way, our one-dimensional sorts could be used as subcomponents of these same algorithms for application on a two-dimensional sub-bus array.

There are a number of architectures related to the sub-bus mesh array, namely, the mesh array with fixed buses and the many flavors of reconfigurable mesh. A mesh array with fixed buses is a conventional mesh of processors with nearest neighbor communication links that has been augmented with buses. These buses connect rows of processors along each dimension and cannot be segmented into sub-buses (see, for example, [13], [11]). A reconfigurable mesh is a general term describing mesh architectures where each processor can dynamically configure connections with its nearest neighbors. The connections give rise to buses connecting arbitrary subsets of processors in the mesh. (e.g., see [12]). With only a constant factor of overhead, the reconfigurable meshes can simulate a sub-bus mesh, the sub-bus mesh can simulate a mesh with fixed buses, and a mesh with buses can simulate a conventional mesh. One-dimensional versions of the sub-bus array and the reconfigurable arrays are essentially equivalent.

Rajasekaran [14] provides a summary and pointers to many of the sorting results on fixed bus and reconfigurable mesh architectures, as well as results for the related problem of routing. Most of the algorithms from this body of work assume queue sizes larger than one or involve sorting more than one item per processor. These cannot be directly applied to our sorting models since we focus on the problem of sorting in place.

We have found an interesting relationship between our greedy sorting strategies and sequential sorting algorithms: the permutation that results from the application of a left greedy insertion step is the same as what would result from applying one pass of the sequential sorting algorithm *bubble sort* [9]. The number of passes used by bubble sort to sort a permutation is exactly the number of insertion steps used by the left greedy sorting strategy. In fact, our average-case analysis of the left greedy sorting strategy is an alternative to the analysis of bubble sort given by Demuth [4] and Knuth [9]. A common improvement of bubble sort is to alternate the directions of the sorting passes over the input. This "cocktail shaker sort" has been observed to use about half as many sorting passes as bubble sort. Our analysis and simulation of alternating greedy sort gives evidence as to why this is the case. We discuss the parallel insertion model's connections with bubble sort in section 5.

In our preliminary paper [6], we defined the left adaptive insertion step which is closely related to our left simple insertion step. We proved that the left adaptive sort is an optimal left-only sorting strategy using arguments similar to theorem 3.5 and lemma 3.2 for the left simple insertion sorting strategy. The left adaptive insertion step can be implemented in four sub-bus operations whereas the left simple insertion step can be implemented in just three sub-bus operations, making the latter's implementation slightly faster in practice.

## 2 The Parallel Insertion Model

We begin by defining a simple abstract model for sub-bus sorting, which we call the *parallel insertion model*, for one-dimensional, in-place, comparison sub-bus sorting. In the parallel insertion model the data to be sorted is represented by a permutation $\pi$ of $\{1, 2, ..., n\}$ where $n$ is the number of processors. The data value $\pi[i]$ is stored at processor $i$; $\pi$ is considered sorted if $\pi[i] = i$ for $1 \leq i \leq n$. To simplify our definitions we assume that there are dummy processors 0 and $n+1$ with $\pi[0] = -\infty$ and $\pi[n+1] = \infty$. A *left insertion step* $\beta$ is defined by a set of *active* processors $A_\beta \subseteq \{0, 1, 2, ..., n\}$, where 0 is always a member of $A_\beta$. The permutation $\beta(\pi)$ is defined to be

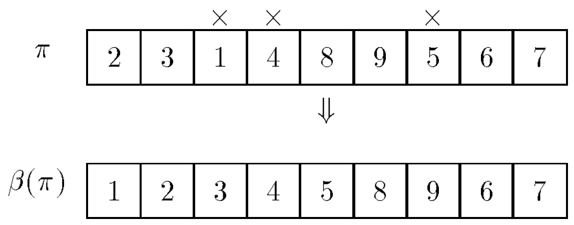

Figure 1: Data movement in a left insertion step $\beta$ applied to $\pi$ where $A_\beta = \{0, 3, 4, 7\}$

the permutation which is the result of moving the data values $\pi[i+1], ..., \pi[j-1]$ to the processors $i+2, i+3, ..., j$, respectively, and value $\pi[j]$ to processor $i+1$ for every pair $i, j$ of consecutive active processors. To be more precise, let $A_\beta = \{i_0, i_1, i_2, \ldots, i_k\}$ with $0 = i_0 < i_1 < i_2 < \ldots < i_k$. The result of the left insertion step $\beta$ applied to $\pi$ is the permutation $\beta(\pi)$ where

$$\beta(\pi)[i] = \begin{cases} \pi[i_j] & \text{if } i = i_{j-1} + 1 \text{ and } 1 \leq j \leq k \\ \pi[i-1] & \text{if } i - 1 \notin A_\beta \text{ and } i - 1 < i_k \\ \pi[i] & \text{if } i \notin A_\beta \text{ and } i > i_k. \end{cases}$$

Figure 1 illustrates a left insertion with active processors 0, 3, 4, and 7. Notice that since consecutive processors 3 and 4 are both active, the value of 4 does not move. In a symmetric way we can define *right insertion steps*. For a right insertion step $\beta$, $A_\beta$ we have $A_\beta \subseteq \{1, 2, ..., n, n+1\}$, where $n+1$ is always a member of $A_\beta$. A *sorting strategy for* $\pi$ is a sequence of insertion steps $\beta_1, \beta_2, ...\beta_T$ where $\pi = \pi_0$, for $1 \leq i \leq T$, $\pi_i = \beta_i(\pi_{i-1})$, and $\pi_T$ is sorted. We say that the sorting strategy $\beta_1, \beta_2, ...\beta_T$ sorts $\pi$ in $T$ steps.

A *left-only sorting strategy* for $\pi$ is a sorting strategy for $\pi$ that consists entirely of left insertion steps. Similarly, a *right-only sorting strategy* uses only right insertion steps. A *one-way sorting strategy* is one which is either a left-only or right-only sorting strategy. A more general sorting strategy is a *two-way sorting strategy* which allows both left and right insertion steps. A special case of a two-way sorting strategy is an *alternating sorting strategy* where left insertion steps alternate with right insertion steps, starting with a left insertion step and ending with a right insertion step.

An important thing to note about the parallel insertion model is that an insertion step is defined entirely by an active set of processors and an insertion step direction. We place no limitations on how the active set and the insertion step direction is determined. When devising sorting strategies, it is natural to define the insertion steps in an algorithmic way, defining the next insertion step of the strategy based on the permutation that resulted from the previous insertion step. The sorting strategies given below are defined in this manner. The only complexity measure we consider is the number of insertion steps used by the strategy, not how the active sets or the direction of each insertion step was determined. We address the latter complexity issues in a less rigorous way when we discuss the implementation of sorting strategies in section 7.

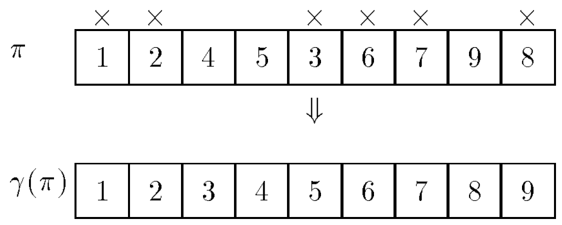

Figure 2: Data movement in a left greedy insertion step.

## 2.1 Greedy Sorting Strategies

A natural approach for devising sorting strategies on the sub-bus is to try to maximize the sorting work performed at each step. The *greedy sorting strategies* are a family of sorting strategies whose insertion steps remove the most inversions of any insertion step. The *set of inversions* in a permutation $\pi$ is defined as the set of out-of-order value pairs, $I(\pi) = \{(x, y) \mid x = \pi[i],\ y = \pi[j] \text{ where } i < j \text{ and } x > y\}$. The set of inversions and the number of inversions of a permutation are useful measures of order within a permutation, and have been applied to the analysis of sequential sorting [9]. The only permutation with no inversions is the sorted permutation. Define the *left greedy insertion step for* $\pi$ as a left insertion step where the set of active processors is

$$G_L(\pi) = \{i \mid 0 \leq i \leq n \text{ and if } i < j \leq n \text{ then } \pi[j] > \pi[i]\}.$$

Note that $n \in G_L(\pi)$ for all $\pi$, vacuously. An example of the action of a left greedy insertion step can be seen in figure 2. The values at active processors form an increasing sequence from left to right, as prescribed by the active set definition.

An important property of a left greedy insertion step is that it never creates any new inversions. This is stated in the following lemma.

**Lemma 2.1** *Let $\gamma$ be the left greedy insertion step for $\pi$. If $i \in G_L(\pi)$ and $\gamma(\pi)[j] = \pi[i]$, then for all $j \leq k < i$, $\pi[i] < \pi[k]$.*

**Proof**: Let $i \in G_L(\pi)$ and $\gamma(\pi)[j] = \pi[i]$, and suppose the lemma were false. There is a maximal $k$ with $j \leq k < i$ and $\pi[i] > \pi[k]$. Then for all $l$ with $k < l < i$, we know that $\pi[l] > \pi[i] > \pi[k]$ by our choice of $k$. Also, since $i \in G_L(\pi)$ we know that for all $l > i$, $\pi[l] > \pi[i]$ as well. Hence, $\pi[l] > \pi[k]$ for all $l > k$. As a consequence $k \in G_L(\pi)$. However, $k \notin G_L(\pi)$ because $\gamma(\pi)[j] = \pi[i]$ and $j \leq k < i$. □

Our next theorem demonstrates that a left greedy insertion step truly exhibits greedy behavior by removing the most inversions of any left insertion step.

**Theorem 2.1** *Let $\gamma$ be the left greedy insertion step for $\pi$. For all possible left insertion steps $\beta$ for $\pi$, $|I(\gamma(\pi))| \leq |I(\beta(\pi))|$.*

**Proof**: Let $\omega$ be a left insertion step with the property that for all left insertion steps $\beta$, $|I(\omega(\pi))| \leq |I(\beta(\pi))|$ and among those with the property $A_\omega$ is as large a set as possible. We will demonstrate that $G_L(\pi) = A_\omega$ which implies the result.

CLAIM 1: $n \in A_\omega$.

Suppose $n \notin A_\omega$. Choose $i$ to be the largest member of $A_\omega$. Define left insertion step $\nu$ by the active set $A_\nu = A_\omega \cup \{i+1, i+2, \ldots, n\}$. It follows that $\nu(\pi) = \omega(\pi)$ but $A_\nu$ has more more members than $A_\omega$, which contradicts our choice of $\omega$.

CLAIM 2: $G_L(\pi) \subseteq A_\omega$.

Suppose otherwise, then choose $i$ to be as large as possible such that $i \in G_L(\pi) - A_\omega$. Let $A_\nu = A_\omega \cup \{i\}$. That is, we form the new left insertion step $\nu$ from $\omega$ by adding $i$ as an active processor. By the previous claim $i \neq n$ and $n \in A_\omega$. Choose $j$ to be as small as possible such that $j > i$ and $j \in A_\omega$. Since $i \in G_L(\pi)$ we must have $\pi[i] < \pi[j]$. Thus, we have $(\pi[j], \pi[i]) \in I(\omega(\pi)) - I(\nu(\pi))$. On the other hand, by the reasoning in the proof of lemma 2.1 making $i$ active in $\nu$ will not introduce any inversions than $\omega$ does not already introduce. Thus, $|I(\nu(\pi))| < |I(\omega(\pi))|$, which is impossible.

CLAIM 3: $A_\omega \subseteq G_L(\pi)$

Suppose otherwise, then choose $i$ to be as large as possible such that $i \in A_\omega - G_L(\pi)$. Let $A_\nu = A_\omega - \{i\}$. That is, we form the new left insertion step $\nu$ from $\omega$ by deleting $i$ as an active processor. Since $n \in G_L(\pi)$, $i \neq n$. Choose $j$ to be as small as possible such that $j > i$ and $j \in G_L(\pi)$. By the previous claim and the choice of $j$, $j$ is also the smallest member of $A_\omega$ larger than $i$. We have $0 \in G_L(\pi)$, so choose $k < i$ to be as large as possible such that $k \in G_L(\pi)$. By the previous claim $k \in A_\omega$. Thus, by lemma 2.1, no new inversions are created by removing $i$ from $A_\omega$, that is, $I(\nu(\pi)) \subseteq I(\omega(\pi))$. On the other hand, the removal of $i$ from $A_\omega$ means that the inversion $(\pi[i], \pi[j]) \in I(\omega(\pi)) - I(\nu(\pi))$. Thus, $|I(\nu(\pi))| < |I(\omega(\pi))|$, which is impossible. □

The *left greedy sorting strategy* for $\pi$ is the left-only sorting strategy for $\pi$ where each insertion step is left greedy. The *alternating greedy sorting strategy* for $\pi$ is the alternating sorting strategy for $\pi$ where each left insertion step is left greedy and each right insertion step is right greedy.

## 2.2 Simple Insertion Sorting Strategies

A second family of sorting strategies that we will consider is the *simple insertion sorting strategies* that are based on *left and right simple insertion steps.* A left simple insertion step is based on the idea that a processor should be active when its value is inverted with the value at the processor to its immediate left. Determining the active processors in a left simple insertion step for $\pi$ is a two step process. First, we define the *pre-active* set of processors $P_L(\pi)$ to be

$$P_L(\pi) = \{i \mid 1 \leq i \leq n \text{ and } \pi[i-1] > \pi[i]\} \cup \{n+1\}.$$

Making only the pre-active processors active is not enough to ensure a sorting step where no inversions are introduced. Figure 3a illustrates a situation in which inversions would be created by a left insertion step with only the pre-active processors being active. The values of 3 and 8 are inserted as far left as possible, creating inversions. To correct this, we extend the active set in the

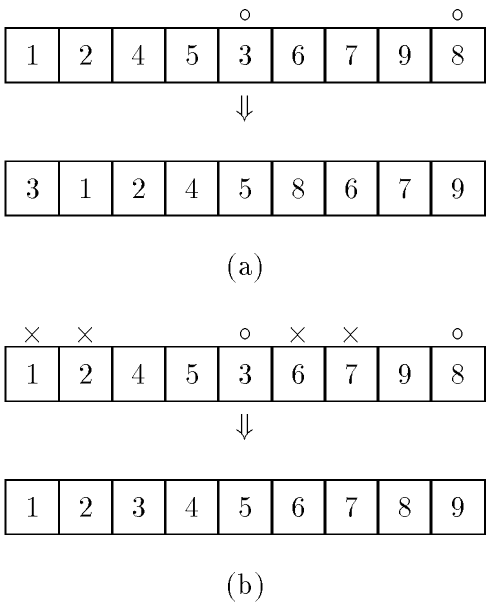

Figure 3: (a) Inversions created by left insertion step with active set $P_L(\pi)$. (b) Corrected left insertion step with additional active processors $B_L(\pi)$.

obvious way: given a permutation $\pi$, define the *blocking* set of processors

$$\begin{aligned} B_L(\pi) \quad = \quad & \{i \mid 0 \leq i \leq n \text{ and for some } j > i, j \in P_L(\pi), \pi[i] < \pi[j], \\ & \text{and for } i \leq l < j, l \notin P_L(\pi)\}. \end{aligned}$$

The additional active processors simply insure that a value at an active processor does not create any inversions with the left insertion step. As we shall see in the proof of lemma 3.2, claim 3, the blocking processors within a segment bounded by two pre-active processors always form a contiguous block at the left end of the segment. These blocking processors stop the value at the pre-active processor to their right from inserting to their left. Since the dummy processor $n+1$ is included in $P_L(\pi)$ and has infinite value, processors at the right end of the array that are not pre-active will be in the blocking set. This simplifies some of our proofs about and implementation of simple insertion sort. Figure 3b illustrates the previous example with the proper blocking processors. The 3 and the 8 are blocked by the values of 2 and 7, respectively, and this is the behavior we desire.

The *left simple insertion step for* $\pi$ is a left insertion step whose set of active processors is the combined set

$$S_L(\pi) = P_L(\pi) \ \cup \ B_L(\pi).$$

We can define a *right simple insertion step for* $\pi$ in a similar way. The *left simple insertion sorting strategy for* $\pi$ is the left-only sorting strategy where each insertion step is a left simple insertion step. The *alternating simple insertion sorting strategy* for $\pi$ is the alternating sorting strategy for $\pi$ where the left insertion steps are left simple and the right insertion steps are right simple.

## 2.3 The Odd-Even Transposition Sorting Strategy

An alternative to the parallel insertion model one might consider is one that only allows exchanges of values (transpositions) rather than insertion operations. Equivalently, in the parallel insertion model you could consider insertion steps where there is at most one consecutive inactive processor. We can then formulate odd-even transposition sort [10, 7] which uses value exchanges as another example of a parallel insertion sorting strategy.

Define the *odd transposition step* for $\pi$ as a left insertion step with the set of active processors

$$T_1(\pi) = \{ \ i \mid 0 \leq i \leq n \text{ and } i \bmod 2 = 0 \text{ or } \pi[i] < \pi[i+1] \ \}.$$

Similarly define the *even transposition step* for $\pi$ as a left insertion step with the set of active processors

$$T_0(\pi) = \{ \ i \mid 0 \leq i \leq n \text{ and } i \bmod 2 = 1 \text{ or } \pi[i] < \pi[i+1] \ \}.$$

Recall that $\pi[0] = -\infty$ and $\pi[n+1] = \infty$, so $0, n \in T_b(\pi)$ for $b \in \{0, 1\}$ and all $\pi$. The *odd-even transposition sorting strategy* for $\pi$ is an alternating sequence of odd and even transposition sorting steps that sorts $\pi$, starting with an odd transposition step. Figure 4 shows the first two steps of the odd-even transposition sorting strategy for a specific $\pi$. Because there is at most one consecutive inactive processor, insertion operations involve value exchanges. Note that odd-even transposition is a left-only sorting strategy since all of the steps are left insertion steps.

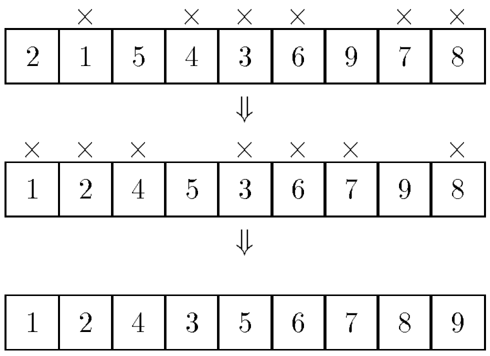

Figure 4: The first two steps of the odd-even transposition sorting strategy for a permutation. The odd transposition step exchanges values at odd-even processor pairs, and the even transposition exchanges values at even-odd processor pairs.

It can be shown that the odd-even transposition sorting strategy will always sort in $n$ insertion steps where $n$ is the permutation size. Typically an *oblivious* version of the sort is implemented which always sorts in $n$ steps regardless of the permutation. We will also consider the *early-stopping* version of the odd-even transposition sorting strategy for $\pi$ which consists of only the steps necessary to sort $\pi$.

## 3 One-Way Sorting Strategies

In this section we consider left-only sorting strategies. Although left-only sorting strategies are less general than two-way sorting strategies, their development and analysis are interesting and help us understand the more efficient two-way sorting strategies. In addition, any lower bounds we find for one-way sorting strategies apply directly to the early stopping version of odd-even transposition sort.

We begin by characterizing the number of steps required by any one-way sorting strategy to sort a given permutation. We then show that the one-way greedy and one-way simple insertion sorting strategies are optimal.

### 3.1 Lower Bounds for Left-Only Sorting Strategies

Left-only sorting strategies are limited by the fact that a value which is left of its final position can move at most one position to the right per left insertion step. Let $1 \leq x \leq n$ and $\pi$ be a permutation of $\{1, 2, \ldots, n\}$. Suppose $i$ is the location of the value $x$ in $\pi$, so $x = \pi[i]$. Then we define $\mathrm{dist}_L(x, \pi) = \max\{0, x - i\}$. Since the final destination of $x$ is processor $x$, $\mathrm{dist}_L(x, \pi)$ is the distance in $\pi$ of $x$ from its final destination if $x$ is to the left of its final destination. Otherwise, $\mathrm{dist}_L(x, \pi) = 0$. Define $\mathrm{maxdist}_L(\pi) = \max_x \ \mathrm{dist}_L(x, \pi)$, so $\mathrm{maxdist}_L(\pi)$ is the maximum distance any value is to the left to its final destination in $\pi$.

We observe that $\pi$ is sorted if and only if $\mathrm{maxdist}_L(\pi) = 0$. Clearly, if $\pi$ is sorted then $\mathrm{maxdist}_L(\pi) = 0$. If $\pi$ is not sorted then let $x$ be as large as possible such that $\pi[x] \neq x$. Let $\pi[i] = x$. Since $\pi[j] = j$ for all $j > x$ then $i < x$. Thus,

$$\mathrm{maxdist}_L(\pi) \geq \mathrm{dist}_L(x, \pi) = \max\{0, x - i\} = x - i > 0.$$

Our first theorem is an elementary lower bound on left-only sorting strategies:

**Theorem 3.1** *If $\beta_1, \beta_2, \ldots \beta_T$ is a left-only sorting strategy for $\pi$, then $T \geq maxdist_L(\pi)$.*

**Proof**: This follows from the observation that if $\pi$ is unsorted and $\beta$ is a left insertion step then $\mathrm{maxdist}_L(\beta(\pi)) \geq \mathrm{maxdist}_L(\pi) - 1$. □

We define $E(n)$ to be the expected value of $\mathrm{maxdist}_L(\pi)$ where each permutation $\pi$ of $\{1, 2, ...n\}$ is equally likely. We have the following theorem:

**Theorem 3.2** *The expected value of $maxdist_L(\pi)$ is*

$$E(n) = n - \frac{1}{n!} \sum_{k=0}^{n} k! k^{n-k}.$$

**Proof**: Define $m_k(n)$ to be the number of permutations $\pi$ of $\{1, 2, ..., n\}$ such that $\text{maxdist}_L(\pi) \leq k$. The function $m_k$ can be expressed recursively as follows:

$$m_k(n) = \begin{cases} n! & \text{if } n \leq k+1 \\ (k+1)m_k(n-1) & \text{if } n > k+1 \end{cases}$$

To see the case where $n > k+1$, there are $k+1$ distinct $i$'s which can be chosen such that $\pi[i] = n$ and $\text{dist}_L(n, \pi) \leq k$. Once $i$ is chosen such that $\pi[i] = n$ and $\text{dist}_L(n, \pi) \leq k$, then the number of ways to choose the remaining components of the permutation $\pi$ satisfying $\text{maxdist}_L(\pi) \leq k$ is simply $m_k(n-1)$. Unwinding the recursion we obtain $m_k(n) = (k+1)!(k+1)^{n-k-1}$ for $n \geq k+1$. Thus,

$$\begin{aligned} E(n) &= \frac{1}{n!}\sum_{k=1}^{n-1} k[m_k(n) - m_{k-1}(n)] \\ &= \frac{1}{n!}[nm_{n-1}(n) - \sum_{k=0}^{n-1} m_k(n)] \\ &= n - \frac{1}{n!}\sum_{k=0}^{n-1}(k+1)!(k+1)^{n-k-1} \\ &= n - \frac{1}{n!}\sum_{k=0}^{n} k!k^{n-k}. \end{aligned}$$

□

As it happens, the expression $\frac{1}{n!}\sum_{k=0}^{n} k!k^{n-k}$ can be approximated very closely as $\sqrt{\frac{\pi n}{2}} + \frac{2}{3} - \Theta(n^{-\frac{1}{2}})$ (cf. Knuth Vol. 1, pages 112-117 [8]). Combining this fact with the previous two theorems we have:

**Theorem 3.3** *The expected number of steps in any left-only sorting strategy is at least*

$$n - \sqrt{\frac{\pi n}{2}} + \frac{2}{3} - \Theta(n^{-\frac{1}{2}}).$$

Thus, the advantage of a left-only sorting strategy over oblivious odd-even transposition sort can only be slight. Nonetheless, it is interesting to see if optimality for left-only sorting strategies can be achieved.

## 3.2 Optimality of the Left Greedy Sorting Strategy

The left greedy sorting strategy is an optimal left-only sorting strategy as evidenced by the following theorem.

**Theorem 3.4** *If $\gamma_1, \gamma_2, \ldots, \gamma_T$ is the left greedy sorting strategy for $\pi$, then $T = maxdist_L(\pi)$.*

**Proof**: By the lower bound theorem, to prove optimality of $\gamma_1, \gamma_2, \ldots, \gamma_T$ with respect to $\pi$ we have to show $T \leq \mathrm{maxdist}_L(\pi)$. To do this we simply have to show that each left greedy insertion step reduces the maximum distance left of values in $\pi$ by one. That is, for $1 \leq i \leq T$, $\mathrm{maxdist}_L(\pi_i) = \mathrm{maxdist}_L(\pi_{i-1}) - 1$, where $\pi_0 = \pi$ and $\pi_i = \gamma_i(\pi_{i-1})$. This follows from lemma 3.1 below. □

**Lemma 3.1** *If $\pi$ be an unsorted permutation and $\gamma$ is the greedy insertion step for $\pi$, then $maxdist_L(\gamma(\pi)) = maxdist_L(\pi) - 1$.*

**Proof**: The proof of the lemma is done by two claims. The first claim states that if the value at any inactive processor is left of its final destination then it moves one processor closer to its final destination as the result of the left greedy insertion step. The second claim states that the value at any active processor is at or to the right of its final destination.

The following formalizes these claims. Let $i$ be a processor with $x = \pi[i]$ and $1 \leq i \leq n$.

Claim 1: If $i \notin G_L(\pi)$ then either $\mathrm{dist}_L(x, \gamma(\pi)) = \mathrm{dist}_L(x, \pi) = 0$ or $\mathrm{dist}_L(x, \gamma(\pi)) = \mathrm{dist}_L(x, \pi) - 1$.

Suppose $i \notin G_L(\pi)$. Since $n \in G_L(\pi)$, choose $j$ as small as possible such that $j > i$ and $j \in G_L(\pi)$. By definition of left insertion steps, $\gamma(\pi)[i+1] = x$. If $\mathrm{dist}_L(x, \pi) = 0$ then $x - i \leq 0$, so $x - i - 1 < 0$. Hence, $\mathrm{dist}_L(x, \gamma(\pi)) = \max\{0, x - i - 1\} = 0$. If $\mathrm{dist}_L(x, \pi) > 0$ we have $x - i > 0$, so $x - i - 1 \geq 0$. It follows that $\mathrm{dist}_L(x, \gamma(\pi)) = \max\{0, x - i - 1\} = \mathrm{dist}_L(x, \pi) - 1$.

Claim 2: If $i \in G_L(\pi)$ then $\mathrm{dist}_L(x, \gamma(\pi)) = \mathrm{dist}_L(x, \pi) = 0$.

Suppose $i \in G_L(\pi)$. Then for all $j > i$ we have $x < \pi[j]$. Let $\gamma(\pi)[k] = x$, so $k$ is the new location of $x$ after step $\gamma$. By lemma 2.1, for all $j$ with $k < j \leq i$, $\gamma(\pi)[j] = \pi[j-1] > x$. Also, for all $j > i$, there is some $l > i$ with $\gamma(\pi)[j] = \pi[l] > x$. That is, all the values in positions greater than $i$ in $\pi$ stay in positions greater than $i$ in $\gamma(\pi)$. Thus, for all $j > k$, $\gamma(\pi)[j] > x$. There are $n - k$ values larger than $x$, so $x \leq n - (n - k) = k$ which means $\mathrm{dist}_L(x, \gamma(\pi)) = \max\{0, x - k\} = 0$.

Having established these claims, the lemma follows easily:

$$\begin{aligned}
\mathrm{maxdist}_L(\gamma(\pi)) &= \max_x \mathrm{dist}_L(x, \gamma(\pi)) \\
&= \max\{\mathrm{dist}_L(x, \gamma(\pi)) \mid x = \pi[i] \text{ and } i \notin G_L(\pi)\} \\
&= \mathrm{maxdist}_L(\pi) - 1
\end{aligned}$$

The first equality follows from the definition of $\mathrm{maxdist}_L$, the second equality from claim 2, and the last equality from claim 1. □

## 3.3 Optimality of the Left Simple Insertion Sorting Strategy

We now prove the optimality of the left simple insertion sorting strategy as stated in the theorem below.

**Theorem 3.5** *If $\sigma_1, \sigma_2, \ldots, \sigma_T$ is the left simple insertion sorting strategy for a permutation $\pi$, then $T = maxdist_L(\pi)$.*

**Proof**: Let $\pi$ be a permutation and $\sigma_1, \sigma_2, \ldots, \sigma_T$ be the left simple insertion sorting strategy for $\pi$. By the lower bound theorem, to prove optimality we have to show $T \leq \mathrm{maxdist}_L(\pi)$. To do this we simply have to show that each left simple insertion step reduces the maximum distance left of values in $\pi$ by one. That is, for $1 \leq i \leq T$, $\mathrm{maxdist}_L(\pi_i) = \mathrm{maxdist}_L(\pi_{i-1}) - 1$, where $\pi_0 = \pi$ and $\pi_i = \sigma_i(\pi_{i-1})$. This follows from lemma 3.2 below. □

**Lemma 3.2** *Let $\pi$ be an unsorted permutation and $\sigma$ be the left simple insertion step for $\pi$. Then $maxdist_L(\sigma(\pi)) = maxdist_L(\pi) - 1$.*

**Proof**: The proof of the lemma is done by a series of three claims. The first claim states that if the value at any inactive processor is left of its final destination then it moves one processor closer to its final destination as the result of the left simple insertion step. The second claim states that the value at a pre-active processor after the left insertion step is never further left of its final destination than the value at some inactive processor. The third claim states that the value at a blocking processor after the left simple insertion step is never further left of its final destination than the value at some pre-active processor. The cascading effect of these three claims is that the maximum distance left of the values must decrease by one as a result of the left simple insertion step.

The following formalizes these claims. Let $i$ be a processor with $x = \pi[i]$ and $1 \leq i \leq n$.

Claim 1: If $i \notin S_L(\pi)$ then $\mathrm{dist}_L(x, \sigma(\pi)) = \mathrm{dist}_L(x, \pi) = 0$ or $\mathrm{dist}_L(x, \sigma(\pi)) = \mathrm{dist}_L(x, \pi) - 1$.

Suppose $i \notin S_L(\pi)$. Since $n + 1 \in S_L(\pi)$, there is a $j \in S_L(\pi)$ with $j > i$ and for $i < l < j$, $l \notin S_L(\pi)$. By the definition of left insertion steps we have $\sigma(\pi)[i + 1] = x$. Thus, either $\mathrm{dist}_L(x, \sigma(\pi)) = \mathrm{dist}_L(x, \pi) = 0$ or $\mathrm{dist}_L(x, \sigma(\pi)) = \mathrm{dist}_L(x, \pi) - 1$.

Claim 2: If $i \in P_L(\pi)$ then there exists a $j \notin S_L(\pi)$ with $y = \pi[j]$ and $\mathrm{dist}_L(x,\sigma(\pi)) \leq \mathrm{dist}_L(y, \sigma(\pi))$.

We show this by looking at the contiguous block of pre-active processors that contains $i$. There are two cases to consider: $i$ is the processor at the left end of this pre-active block or $i$ has a pre-active processor to its left in this pre-active block.

Case 1: $i - 1 \notin P_L(\pi)$.

Define $j$ by $\sigma(\pi)[j] = \pi[i]$. Since $i \in P_L(\pi)$, $\pi[i - 1] > \pi[i]$. Thus $i - 1 \notin S_L(\pi)$ and $i - 1 > 0$. It follows that $j < i$ and $j \notin S_L(\pi)$.

Let $y = \pi[j]$. Observe that $y > x$, otherwise $j \in B_L(\pi)$. We can restate this observation as $y \geq x + 1$. Note that $\sigma(\pi)[j + 1] = y$ so we have

$$\mathrm{dist}_L(x, \sigma(\pi)) = \max\{0, x - j\}$$

$$\begin{aligned} &= \max\{0, (x+1)-(j+1)\} \\ &\leq \max\{0, y-(j+1)\} \\ &= \mathrm{dist}_L(y, \sigma(\pi)). \end{aligned}$$

Therefore, with this choice of $j \notin S_L(\pi)$ and $y = \pi[j]$, we have $\mathrm{dist}_L(x, \sigma(\pi)) \leq \mathrm{dist}_L(y, \sigma(\pi))$.

Case 2: $i - 1 \in P_L(\pi)$.

Because $i - 1 \in P_L(\pi)$, we know that $\sigma(\pi)[i] = x$.

Let $k < i$ be the maximal element of $P_L(\pi)$ with $k - 1 \notin P_L(\pi)$. By our choice of $k$, for all $l$ with $k < l \leq i$ we have $l \in P_L(\pi)$. So for all such $l$, $\pi[l-1] > \pi[l]$. The values at processors $k$ to $i$ form a decreasing sequence from left to right. Let $z = \pi[k]$. We can conclude by transitivity that $z > x$.

The key fact about $k$ is that by definition $k \in P_L(\pi)$ *and* $k - 1 \notin P_L(\pi)$, so its value falls under Case 1 above. By that same logic, we know there is an $j \notin S_L(\pi)$ with $y = \pi[j]$ and $\mathrm{dist}_L(z, \sigma(\pi)) \leq \mathrm{dist}_L(y, \sigma(\pi))$. Since $k \in P_L(\pi)$, $\mathrm{dist}_L(z, \pi)) \leq \mathrm{dist}_L(z, \sigma(\pi))$.

Combining these observations, it follows that

$$\begin{aligned} \mathrm{dist}_L(x, \sigma(\pi)) &= \max\{0, x-i\} \\ &\leq \max\{0, z-i\} \\ &\leq \max\{0, z-k\} \\ &= \mathrm{dist}_L(z, \pi) \\ &\leq \mathrm{dist}_L(z, \sigma(\pi)) \\ &\leq \mathrm{dist}_L(y, \sigma(\pi)). \end{aligned}$$

Again, for this choice of $j \notin S_L(\pi)$, we have $\mathrm{dist}_L(x, \sigma(\pi)) \leq \mathrm{dist}_L(y, \sigma(\pi))$.

Case 2 of Claim 2 is illustrated in figure 5. There is a contiguous block of pre-active processors starting at processor $k$ and including $i$ with values decreasing from left to right. The left distance of $x$ does not change with $\sigma(\pi)$, and is bounded by the left distance of $z$ since $z > x$. The value $z$ on the left end of the block moves left past $y$. For $z$ to be farther left of its destination than $y$, $z$ would have to be greater than $y$. But $z < y$ instead so $z$'s distance left is bounded by the left distance of $y$. Note that if we let $k = i$ and $z = x$ this figure illustrates Case 1 as well.

Claim 3: If $i \in B_L(\pi)$ then there is a $j \in P_L(\pi)$ with $y = \pi[j]$ and $\mathrm{dist}_L(x, \sigma(\pi)) \leq \mathrm{dist}_L(y, \sigma(\pi))$.

The proof of this case is similar to the proof of Case 2. Here we examine the values at the contiguous block of processors in $B_L(\pi)$ that contains $i$. It turns out that these blocking processors' left distances are bounded by the left distance of the value at the pre-active processor that they block. Suppose $i \in B_L(\pi)$. By definition of $B_L(\pi)$ there is a $j \in P_L(\pi)$ with $j > i$ and $y = \pi[j]$ such that $x < y$ and for $i \leq l < j$, $l \notin P_L(\pi)$. This implies that for all $i \leq l < j$, $\pi[l-1] < \pi[l]$. Also, since $i \notin P_L(\pi)$, $\pi[i-1] < \pi[i] < y$. Thus either $i - 1 \in P_L(\pi)$ or $i - 1 \in B_L(\pi)$, and so $\sigma(\pi)[i] = x$.

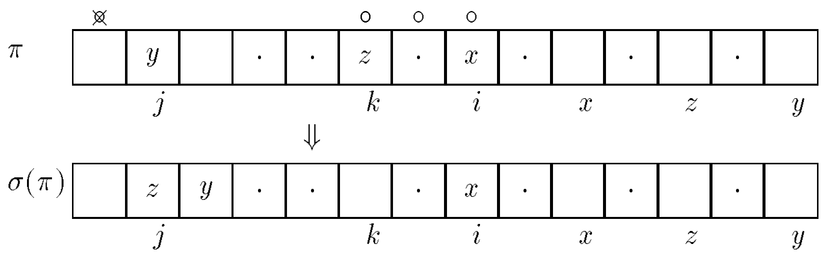

Figure 5: Data movement of $x$, $y$, and $z$ with left simple insertion step $\sigma$ described in Case 2 of Claim 2.

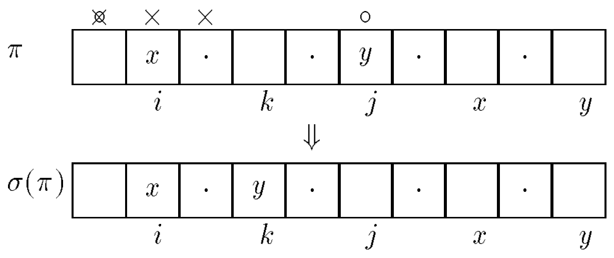

Figure 6: Data movement of $x$ and $y$ with left simple insertion step $\sigma$ described in Claim 3.

Define $k$ by $\sigma(\pi)[k] = y$. This means that $k-1 \in S_L(\pi)$ and, from above, $k-1 \notin P_L(\pi)$. Thus $k-1 \in B_L(\pi)$ and $\pi[k-1] < y$. We know that for all $i \leq l < k$, $\pi[l-1] < \pi[l]$. Thus, there are $k-i$ values $l$ in the interval $i \leq l < k$ with $\pi[l] < y$ and $x \leq \pi[l]$. Hence $x + k - i \leq y$.

With this fact, we can finish the proof of the final case:

$$\begin{aligned}
\text{dist}_L(x, \sigma(\pi)) &= \max\{0, x-i\} \\
&= \max\{0, (x+k-i)-k\} \\
&\leq \max\{0, y-k\} \\
&= \text{dist}_L(y, \sigma(\pi)).
\end{aligned}$$

Thus there exists a $j \in P_L(\pi)$ with $y = \pi[j]$ with $\text{dist}_L(x, \sigma(\pi)) \leq \text{dist}_L(y, \sigma(\pi))$.

This case is shown in figure 6. The situation is very similar to that shown in figure 5 for the previous case, except for the run of blocking actives between $i$ and $k$.

Having established these three claims and using the fact that $S_L(\pi) = P_L(\pi) \cup B_L(\pi)$, the lemma follows easily:

$$\begin{aligned}
\text{maxdist}_L(\sigma(\pi)) &= \max_x \text{dist}_L(x, \sigma(\pi)) \\
&= \max\{\text{dist}_L(x, \sigma(\pi)) \mid x = \pi[i] \ \text{ with } i \in P_L(\pi) \text{ or } i \notin S_L(\pi)\} \\
&= \max\{\text{dist}_L(x, \sigma(\pi)) \mid x = \pi[i] \ \text{ with } i \notin S_L(\pi)\} \\
&= \text{maxdist}_L(\pi) - 1
\end{aligned}$$

The first equality follows from the definition of $\mathrm{maxdist}_L$, the second equality follows from claim 3, the third equality from claim 2, and the last equality from claim 1. Thus, the maximum distance left of the permutation decreases by one with each application of a left simple insertion step. □

## 4 Two-way Parallel Insertion Sorting

One-way sorting is an unnatural restriction for studying sub-bus sorting. Considering one-way sorting is mostly useful for showing the limits of the odd-even transposition sorting strategy. In general, sorting strategies could have any combination of left and right insertion steps, and the insertion direction pattern could vary based on the permutation. We now consider the general case of two-way sorting strategies. The distance that a value must travel is no longer a factor in two-way sorting. However, considering the number of values that need to travel across a cut of the array leads to lower bounds for two-way sorting.

### 4.1 Lower Bounds for Parallel Insertion Sorting

Our goal is to find a metric on permutations that characterizes the limitations of sorting in the parallel insertion model, much like our one-way sorting analysis. Consider a bandwidth argument like the following: suppose a permutation $\pi$ has $k$ items in the left half of $\pi$ that need to move to the right half of $\pi$ to be in sorted order. There must also be $k$ items on the right half of $\pi$ that need to move to the left half. With each insertion step, regardless of direction, at most two of these values can move across the middle of $\pi$ (one broadcast and one shifted). Thus after the step there must be at least $k-1$ items on the left that should be on the right, and vice versa. Extending this argument we can say that it will take at least $k$ steps to sort $\pi$. We formalize these notions below.

Given $n$ processors define a *cut* $i$ to be any $i$ with $1 \leq i < n$. This corresponds to a partition of the processors into $\{1, 2, \ldots, i\}$ and $\{i+1, i+2, \ldots, n\}$. Given a permutation $\pi$ and cut $i$ define

$$\mathrm{offset}_L(i, \pi) = \{\pi[j] \mid j \leq i \text{ and } \pi[j] > i\}.$$

The set $\mathrm{offset}_L(i, \pi)$ contains values $x$ which are on the left side of the cut $i$, but should be on the right, that is, the values on the left that are "off-side" of cut $i$. Similarly, define

$$\mathrm{offset}_R(i, \pi) = \{\pi[j] \mid j > i \text{ and } \pi[j] \leq i\}.$$

It should be clear that $\mathrm{offset}_L(i, \pi)$ and $\mathrm{offset}_R(i, \pi)$ are disjoint and that their sizes are equal. Define $\mathrm{off}(i, \pi) = |\mathrm{offset}_L(i, \pi)|$ and $\mathrm{maxoff}(\pi) = \max_i \mathrm{off}(i, \pi)$. Observe that if $\pi$ is sorted, $\mathrm{maxoff}(\pi) = 0$.

In our proofs, we compare the contents of the off-side sets of a cut before and after the application of an insertion step. We say that a value $x = \pi[j]$ *crosses* cut $i$ with step $\beta$ when $x = \beta(\pi)[k]$ and where one of $j$ and $k$ belongs to the set $\{1, 2, \ldots, i\}$ and the other belongs to $\{i+1, i+2, \ldots, n\}$. In a left insertion step where $i$ is not active and there is an active processor somewhere right of $i$, exactly two values cross $i$: the value at the active value that is inserted to the left of cut $i$ and the value at $i$ that shifts right to $i+1$. In a left insertion step where processor $i$ is active, no value crosses the cut $i$. Similar things can be said of right insertion steps, conditional instead on the state of processor $i+1$. Because of these observations, the contents of $\text{offset}_L(i, \beta(\pi))$ and $\text{offset}_R(i, \beta(\pi))$, and the value of $\text{off}(i, \beta(\pi))$ can be determined by considering only $\text{offset}_L(i, \pi)$ and $\text{offset}_R(i, \pi)$ and the values that cross cut $i$ with step $\beta$.

Consider a left insertion step $\beta$ applied to $\pi$ where $i \notin A_\beta$ and where $x = \pi[i]$, and $y = \pi[j]$ cross cut $i$ with $\beta$. If $x > i$ and $y < i$ then we know that $x \in \text{offset}_L(i, \pi)$ and $y \in \text{offset}_R(i, \pi)$. After the application of $\pi$, however, $x \notin \text{offset}_L(i, \pi)$ and $y \notin \text{offset}_R(i, \pi)$ and so $\text{off}(i, \beta(\pi)) = \text{off}(i, \pi) - 1$. Note that this is the best we can do to reduce size of the off-side sets with a left or a right insertion step: if either $x < i$ or $y > i$, then $\text{off}(i, \beta(\pi)) = \text{off}(i, \pi)$; if both $x < i$ and $y > i$, then $\text{off}(i, \beta(\pi)) = \text{off}(i, \pi) + 1$.

Using this new terminology and applying our observations, we have the following lower bound for any sorting strategy for $\pi$:

**Theorem 4.1** *If $\beta_1, \beta_2, ...\beta_T$ is a sorting strategy for $\pi$, then $T \geq$ maxoff$(\pi)$.*

**Proof**: This follows directly from our observations above: for any insertion step $\beta$, permutation $\pi$, and cut $i$, $\text{off}(i, \beta(\pi)) \geq \text{off}(i, \pi) - 1$. Thus for $1 \leq t < T$, $\text{maxoff}(\pi_{t+1}) \geq \text{maxoff}(\pi_t) - 1$ and so $T \geq \text{maxoff}(\pi)$ steps are required to sort $\pi$. □

Define $M(n)$ to be the expected value of $\text{maxoff}(\pi)$ where $\pi$ is chosen uniformly from the permutations of $\{1, 2, \ldots, n\}$. We have the following:

**Theorem 4.2** *The expected value of maxoff$(\pi)$ is $M(n) \geq \lfloor n/2 \rfloor / 2$.*

**Proof**: Define $H(n)$ to be the expected value of $\text{off}(\lfloor n/2 \rfloor, \pi)$ where $\pi$ is chosen uniformly from the permutations of $\{1, 2, \ldots, n\}$. Clearly $\text{maxoff}(\pi) \geq \text{off}(\lfloor n/2 \rfloor, \pi)$ for all $\pi$, so $M(n) \geq H(n)$.

To determine $H(n)$, note that the probability that $\text{off}(\lfloor n/2 \rfloor, \pi) = k$ is just

$$\frac{\binom{\lfloor n/2 \rfloor}{k}\binom{\lceil n/2 \rceil}{\lceil n/2 \rceil - k}}{\binom{n}{\lceil n/2 \rceil}}$$

because this is just the fraction of ways of choosing the right $\lceil n/2 \rceil$ elements of $\pi$ where exactly $k$ of them are less than or equal to $\lfloor n/2 \rfloor$. So $\text{off}(\lfloor n/2 \rfloor, \pi)$ has a hypergeometric distribution [3] whose mean is

$$H(n) = \frac{\lceil n/2 \rceil \lfloor n/2 \rfloor}{n}.$$

Since $M(n) \geq H(n)$ the statement of the theorem follows. □

We can use this bound in conjunction with theorem 4.1 to say the following about the expected number of steps required by sorting strategies

**Theorem 4.3** *The expected number of steps for any sorting strategy is at least $\lfloor n/2 \rfloor / 2$.*

## 4.2 Alternating Sorting Strategies

Recall that an alternating sorting strategy sorts a permutation $\pi$ by applying left and right insertion steps in alternation, starting with a left insertion step and ending with a right insertion step. For alternating sorting strategies we can make a stronger claim:

**Theorem 4.4** *Let $\beta_1, \beta_2, ...\beta_T$ be an alternating sorting strategy for $\pi$. Then $T \geq 2\,maxoff(\pi)$.*

To establish this, we have the following lemma:

**Lemma 4.1** *If $\beta_L$ and $\beta_R$ are left and right insertion steps respectively, then $maxoff(\beta_R\beta_L(\pi)) \geq maxoff(\pi) - 1$.*

**Proof**: If $\text{maxoff}(\beta_L(\pi)) \geq \text{maxoff}(\pi)$ we are done since $\text{maxoff}(\beta_R\beta_L(\pi)) \geq \text{maxoff}(\beta_R(\pi)) - 1$ . Suppose instead that $\text{maxoff}(\beta_L(\pi)) = \text{maxoff}(\pi) - 1$ and let $i$ be a cut with $\text{off}(i, \pi) = \text{maxoff}(\pi)$. It must be true that $i \notin A_{\beta_L}$. Also, two values must cross cut $i$ with $\beta_L$, with one of those values being $\pi[i] \in \text{offset}_L(i, \pi)$. If no values cross $i$ with $\beta_R$, then

$$\text{maxoff}(\beta_R\beta_L(\pi)) \geq \text{off}(i, \beta_R\beta_L(\pi)) = \text{off}(i, \beta_L(\pi)) = \text{maxoff}(\pi) - 1.$$

Otherwise, if two values cross $i$ with $\beta_R$, then $\pi[i] = \beta_L(\pi)[i+1] = \beta_R\beta_L(\pi)[i]$ would be one of them. But $\pi[i] \in \text{offset}_L(i, \pi)$, so

$$\text{maxoff}(\beta_R\beta_L(\pi)) \geq \text{off}(i, \beta_R\beta_L(\pi)) \geq \text{off}(i, \beta_L(\pi)) = \text{maxoff}(\pi) - 1.$$

□

Lemma 4.1 shows that a pair of alternating insertion steps results in at most two values crossing cut $i$ so we can reduce the off-side sets by at most one. This gives us theorem 4.4 and theorem 4.5 below, lower bounds for alternating sorting strategies analogous to theorems 4.1 and 4.3.

**Theorem 4.5** *The expected number of steps in any alternating sorting strategy is at least $\lfloor n/2 \rfloor$.*

## 4.3 Optimality of the Alternating Greedy Sorting Strategy

Theorem 4.4 from the previous section tells us that an alternating sorting strategy must sort in exactly $2\,\text{maxoff}(\pi)$ steps for it to be considered an optimal alternating sorting strategy. Consider the alternating sorting strategy consisting entirely of left and right greedy insertion steps. This alternating greedy sorting strategy is an optimal alternating sorting strategy by the following:

**Theorem 4.6** *If $\gamma_1, \gamma_2, \ldots, \gamma_T$ is the alternating greedy sorting strategy for $\pi$ then $T = 2\,maxoff(\pi)$.*

**Proof**: Let $\pi$ be any permutation, $\gamma_L$ be the left greedy insertion step for $\pi$. and $\gamma_R$ be the right greedy insertion step for $\gamma_L(\pi)$. We will show that for any cut $i$, $\text{off}(i, \gamma_R\gamma_L(\pi)) = \text{off}(i, \pi) - 1$ or

$\text{off}(i,\pi) = 0$. First, we need to make a few observations about the values that cross cut $i$ with the following claim:

Claim L: For any $\pi$, if $\gamma_L$ is the left greedy insertion step for $\pi$ and $i \notin G_L(\pi)$ then there exists a $j > i$ and $y = \pi[j]$ where $y$ crosses cut $i$ with $\gamma_L$ and $y \leq i$.

To see that this is true, note that $n \in G_L(\pi)$ by definition of $G_L(\pi)$. Since $i \notin G_L(\pi)$ there must be a $j \in G_L(\pi)$ with $j > i$ where $y = \pi[j]$ crosses cut $i$ with $\gamma_L$. Since $j \in G_L(\pi)$ we know that $y < \pi[l]$ for all $l > j$. Let $k$ be such that $\gamma_L(\pi)[k] = y$ and note that $k \leq i$. Since $\gamma_L$ creates no inversions, we know that $y < \pi[l]$ for $k \leq l < j$. Therefore $y \leq i$.

We can make a companion claim for right greedy insertion steps:

Claim R: For any $\pi$, if $\gamma_R$ is the right greedy insertion step for $\pi$ and $i+1 \notin G_R(\pi)$ then there exists a $k \leq i$ and $z = \pi[k]$ where $z$ crosses cut $i$ with $\gamma_R$ and $z > i$.

Given these claims we can consider the following four cases:

Case 1: $i \notin G_L(\pi)$, $i+1 \notin G_R(\gamma_L(\pi))$.

Using claim L we know there exists a $y = \pi[j]$ with $j > i$ and $y \leq i$ that crosses cut $i$ with $\gamma_L$. Similarly, from claim R there exists a $z = \gamma_L(\pi)[k]$ with $k \leq i$ and $z > i$ that crosses cut $i$ with $\gamma_R$. Note also that $x = \pi[i]$ crosses cut $i$ with both $\gamma_L$ and $\gamma_R$ because $\gamma_L(\pi)[i+1] = x$ and $\gamma_R\gamma_L(\pi)[i] = x$. So the net effect of $\gamma_L$ and $\gamma_R$ is that only $y$ and $z$ cross cut $i$. Since $y \leq i$ and $z > i$, we have $\text{offset}_R(i, \gamma_R\gamma_L(\pi)) = \text{offset}_R(i,\pi) - \{y\}$ and $\text{offset}_L(i,\gamma_R\gamma_L(\pi)) = \text{offset}_L(i,\pi) - \{z\}$. It follows that $\text{off}(i,\gamma_R\gamma_L(\pi)) = \text{off}(i,\pi) - 1$.

Case 2: $i \notin G_L(\pi)$, $i+1 \in G_R(\gamma_L(\pi))$.

From claim L there exists a $y = \pi[j]$ with $j > i$ and $y \leq i$ that crosses cut $i$ with $\gamma_L$. In addition, the value $x = \pi[i]$ crosses cut $i$ with $\gamma_L$. Because $x = \gamma_L(\pi)[i+1]$ and $i+1 \in G_R(\gamma_L(\pi))$, we know that $x > \gamma_L(\pi)[l]$ for $l < i+1$. Thus $x > i$.

Since $y \leq i$ and $x > i$, we have $\text{offset}_R(i,\gamma_L(\pi)) = \text{offset}_R(i,\pi) - \{y\}$ and $\text{offset}_L(i,\gamma_L(\pi)) = \text{offset}_L(i,\pi) - \{x\}$. It follows that $\text{off}(i,\gamma_L(\pi)) = \text{off}(i,\pi) - 1$. Because $i+1 \in G_R(\gamma_L(\pi))$, no values cross $i$ with $\gamma_R$, and so $\text{off}(i,\gamma_R\gamma_L(\pi)) = \text{off}(i,\gamma_L(\pi)) = \text{off}(i,\pi) - 1$.

Case 3: $i \in G_L(\pi)$, $i+1 \notin G_R(\gamma_L(\pi))$.

Because $i \in G_L(\pi)$, no values cross $i$ with $\gamma_L$, so $\text{off}(i,\gamma_L(\pi)) = \text{off}(i,\pi)$. Let $x = \gamma_L(\pi)[i+1]$ and define $j$ such that $x = \pi[j]$. Note that $j \in G_L(\pi)$ and since $\gamma_L$ creates no inversions that $x < \pi[l]$ for $l > i+1$. From claim R there exists a $z = \gamma_L(\pi)[k]$ with $k \leq i$ and $z > i$ that crosses cut $i$ with $\gamma_R$. Since $\gamma_R$ creates no inversions, it follows that $z > x$. There are at least $n - i$ values $l$ with $x < \pi[l]$, and so $x \leq i$. Since $i+1 \notin G_R(\gamma_L(\pi))$, $x = \gamma_R\gamma_L(\pi)[i]$.

From the preceding argument, we know that $x$ and $z$ cross cut $i$ with $\gamma_R$ and that $x \leq i$ and $z > i$. Therefore, $\text{offset}_R(i, \gamma_L(\pi)) = \text{offset}_R(i, \pi) - \{x\}$ and $\text{offset}_L(i, \gamma_L(\pi)) = \text{offset}_L(i, \pi) - \{z\}$. We conclude that $\text{off}(i, \gamma_R\gamma_L(\pi)) = \text{off}(i, \gamma_L(\pi)) - 1 = \text{off}(i, \pi) - 1$.

Case 4: $i \in G_L(\pi)$, $i + 1 \in G_R(\gamma_L(\pi))$.

No values cross cut $i$ in either step so for all $l > i$, $\pi[l] > \pi[i]$ and for all $l < i + 1$, $\pi[l] < \pi[i]$ by definition of greedy insertion steps. Thus $\text{off}(i, \pi) = 0$.

In all four cases we have either $\text{off}(i, \gamma_R\gamma_L(\pi)) = \text{off}(i, \pi) - 1$ or $\text{off}(i, \pi) = 0$. Thus, if $\pi$ is unsorted we have $\text{maxoff}(\gamma_R\gamma_L(\pi)) = \text{maxoff}(\pi) - 1$, completing the proof. □

The optimality of alternating greedy sort gives us tight bounds on the number of steps required by any alternating sorting strategy. An optimal alternating sorting strategy must reduce the value of maxoff by one every two steps.

## 4.4 Best-Greedy Sorting Strategies

Among all sorting strategies for a permutation $\pi$ that use only greedy insertion steps, there is at least one that uses the least number of greedy insertion steps. Call any greedy strategy that uses the least number of greedy insertion steps to sort $\pi$ a *best-greedy strategy for $\pi$*. The alternating greedy sorting strategy for $\pi$ sorts in at most $2\,\text{maxoff}(\pi)$ steps. Since $2\,\text{maxoff}(\pi) \leq n$, a best-greedy strategy sorts $\pi$ in at most $n$ steps. To find a best-greedy strategy for a permutation $\pi$ we could naively search for a best-greedy strategy for $\pi$ by testing all $2^n$ combinations of left and right greedy insertion steps. There is actually an easier way to find a best-greedy strategy for $\pi$ using the following "commutativity" theorem.

**Theorem 4.7** *For any $\pi$, if step $\gamma_L$ is the left greedy insertion step for $\pi$, $\gamma_R$ is the right greedy insertion step for $\gamma_L(\pi)$, $\gamma'_R$ is the right greedy insertion step for $\pi$, and $\gamma'_L$ is the left greedy insertion step for $\gamma'_R(\pi)$, then $\gamma_R\gamma_L(\pi) = \gamma'_L\gamma'_R(\pi)$.*

As consequence of theorem 4.7 the order that we choose for the directions of a series of greedy insertion steps is unimportant in the sense that all orderings produce the same result. We can find a best-greedy strategy for $\pi$ by applying $l$ left greedy insertion steps followed by $r$ right greedy insertion steps for all $l$ and $r$ where $l + r = 2\,\text{maxoff}(\pi)$. This limits our search to $2\,\text{maxoff}(\pi)$ sorting strategies of length at most $2\,\text{maxoff}(\pi)$. We use this fact to determine a best-greedy strategy for our simulations in section 6.

**Proof:** Let $\pi$ be an arbitrary permutation and let $\gamma_L$, $\gamma_R$, $\gamma'_L$, and $\gamma'_R$ be defined as in the statement of the theorem. Let $1 \leq i \leq n$. If $x = \pi[i]$ then define $i_L$, $i_R$, $i_{RL}$, and $i_{LR}$ to be such that $x = \gamma_L(\pi)[i_L] = \gamma'_R(\pi)[i_R] = \gamma_R(\gamma_L(\pi))[i_{RL}] = \gamma'_L(\gamma'_R(\pi))[i_{LR}]$. That is, $i_L$, $i_R$, $i_{RL}$, and $i_{LR}$ are the positions of $x$ after the applications of $\gamma_L$, $\gamma'_R$, $\gamma_R\gamma_L$, and $\gamma'_L\gamma'_R$, respectively, on the permutation $\pi$. To prove the theorem it suffices to prove the following claim.

CLAIM : For all $x$ and $y$, if $x = \pi[i]$, $y = \pi[j]$ and $i < j$ then $j_{RL} < i_{RL}$ if and only if $j_{LR} < i_{LR}$.

We show that $j_{RL} < i_{RL}$ implies $j_{LR} < i_{LR}$. The other direction can be shown by a symmetric argument.

Assume $j_{RL} < i_{RL}$. Since $i < j$ then it must be the case $y < x$. There are two cases to consider depending on whether or not $j_R < i_R$ . The easy case is when $j_R < i_R$. In this case, since greedy insertion steps do not create inversions then $j_{LR} < i_{LR}$.

We now consider the case when $i_R < j_R$. In order to show that $j_{LR} < i_{LR}$ we must show two facts: (i) $j_R \in G_L(\gamma'_R(\pi))$ and (ii) for all $k$, if $i_R \leq k < j_R$ then $y < \gamma'_R(\pi)[k]$. These two facts will guarantee that the step $\gamma'_L$ inserts $y$ to the left of $x$. We now consider two cases depending on whether or not $j_L < i_L$.

CASE 1: $j_L < i_L$. Since $i < j$ then it must be the case that $y$ is inserted to the left of $x$ in the step $\gamma_L$. In particular, $j \in G_L(\pi)$. This can only happen if for all $k > j$, $y < \pi[k]$. Since $x > y$ and $i < j$, $j \notin G_R(\pi)$. Any element that is inserted from the left of $y$ to the right of $y$ in the step $\gamma'_R$ must be larger than $y$. Hence, $y < \gamma'_R(\pi)[k]$ for all $k > j_R$, which implies $j_R \in G_L(\gamma'_R(\pi))$.

Since $j_L < i_L$ and $i < j$, we have $y < \pi[k]$ for $i \leq k < j$. If $i \in G_R(\pi)$ then those elements to the right of $x$ in $\gamma'_R(\pi)$ are also to the right of $x$ in $\pi$. If $i \notin G_R(\pi)$ then any element that is inserted from the left of $x$ to the right of $x$ in the step $\gamma'_R$ must be larger than $x$ and, consequently, larger than $y$. Hence $y < \gamma'_R(\pi)[k]$ for $i_R \leq k < j_R$.

CASE 2: $i_L < j_L$. Since $j_{RL} < i_{RL}$ it must be the case that $x$ is inserted to the right of $y$ in the step $\gamma_R$. Therefore, for all $k$ where $k < i_L$ or $i_L < k < j_L$, $x > \gamma_L(\pi)[k]$. Since $x > y$ and $j > i$, $i \notin G_L(\pi)$. This means that all the elements to the left of $x$ in $\pi$ are to the left of $x$ in $\gamma_L(\pi)$. Thus for all $k < i$ we have $x > \pi[k]$, and so $i \in G_R(\pi)$. If $j \notin G_L(\pi)$ then all the elements to the left of $y$ in $\pi$ are to the left of $y$ in $\gamma_L(\pi)$. In particular, this means that if $i < k < j$ then $\pi[k] < x$. Hence, if $j \notin G_L(\pi)$ then $x$ is inserted to the right of $y$ in step $\gamma'_R$. This implies that $j_R < i_R$ which is not the case. Hence, we have $j \in G_L(\pi)$. As in case 1, $j \in G_L(\pi)$ implies $j_R \in G_L(\gamma'_R(\pi))$.

Since $i \in G_R(\pi)$ and $i_R < j_R$ then there must be an $m$ such that $i < m < j$, $m \in G_R(\pi)$ and for all $m'$, such that $i < m' < m$, $m' \notin G_R(\pi)$. That is, the element $z = \pi[m]$ blocks $x$ from moving past $y$ in the step $\gamma_R$. In particular, we have $i_R = m - 1$ and $z > x$. Since $j_{RL} < i_{RL}$, $y$ must be inserted to the left of $z$ in step $\gamma_L$, otherwise $z$ would continue to block $x$ in step $\gamma_R$. Thus, $y < \pi[k]$ for $m \leq k < j$. Any element between $x$ and $y$ in $\gamma'_R(\pi)$ is between $z$ and $y$ in $\pi$. Hence $y < \gamma'_R(\pi)[k]$ for $i_R < k < j_R$. Also $y < x = \gamma'_R(\pi)[i_R]$, so we have established that $y < \gamma'_R(\pi)[k]$ for $i_R \leq k < j_R$. □

## 5 Connections with Bubble Sort

There are several significant connections between our parallel insertion model analyses and some of the foundational work in the study of sequential sorting algorithms. As we briefly noted earlier, the sequential sorting algorithm bubble sort is strongly related to our greedy sorting strategies. Suppose that we are sorting a permutation $\pi$ in a standard sequential model. Recall that bubble sort proceeds by making several passes over the permutation, say, from $\pi[n]$ to $\pi[1]$. With each pass of bubble sort, we compare $\pi[i]$ and $\pi[i+1]$, and exchange their values if they are out of order. The value that moves to the left in each exchange is the least value seen during the pass. Thus, the data movement in a single pass of bubble sort is exactly the data movement that would occur by applying a left greedy step to the permutation in the parallel insertion model. It follows that the number of passes used by left greedy sort is exactly that used by bubble sort to sort a permutation $\pi$. That number is $\text{maxdist}_L(\pi)$ as given by theorem 3.4 whose mean value over permutations of size $n$ is given by $E(n)$ in theorem 3.2.

Because of this connection, any analysis of the number of passes made by bubble sort are relevant to the study of one-way sorting in the parallel insertion model. Such analysis was provided in Demuth's thesis from 1956 [4] (a more condensed version of this work appears in [5]) including a derivation of $E(n)$. Knuth [9] gives this analysis in full detail, and also derives the number of sequential sorting operations (exchanges and comparisons) made by bubble sort. For their analyses, Demuth and Knuth considered the number of inversions removed by each pass. For a permutation $\pi$, they consider the number of inversions of $\pi[i]$ with values to its right: the set $I_i(\pi) = \{(\pi[i], \pi[j]) \mid i < j \text{ and } \pi[i] > \pi[j]\}$. For every $\pi[i]$ with non-empty $I_i(\pi)$, one inversion is removed by a pass of bubble sort. It turns out that $|I_i(\pi)| = \text{dist}_L(\pi[i], \pi)$.

Another interesting connection can be found between the parallel insertion model and a simple sequential model for sorting considered by Demuth in his thesis. Demuth's circular, non-reversible memory machine (CNM) consists of a rotating drum of memory cells accessed by a simple processor consisting of a comparator and a single register. Sorting on the CNM is very similar to one-way parallel insertion sorting. Demuth describes a bubble sort-like algorithm that is optimal for this machine, just as left greedy sort is an optimal one-way sorting strategy in the parallel insertion model. Interestingly, left simple insertion sorting strategy can not be easily adapted to the CNM model.

A slightly improved version of bubble sort is often suggested called "cocktail shaker sort", whose sorting passes occur in alternating directions like alternating greedy sort. As an answer to an exercise, Knuth [9](Exercise 9, pages 360-361,663) notes that the number of cocktail shaker sorting passes used to sort $\pi$ is exactly $2\text{maxoff}(\pi) - (0 \text{ or } 1)$. Thus theorem 4.6 is a detailed proof for that exercise. Many have observed that cocktail shaker sort performs in about half the time as bubble sort. In the next section, we verify this observation by experimentally determining the mean value of $\text{maxoff}(\pi)$ and by simulating the alternating greedy sorting strategy.

We discovered the tight connection between the greedy sorting strategies and bubble sort and the connection between our parallel insertion model and the CNM model after we had fully analyzed the greedy sorting strategies. It is fascinating that the classic analysis of sequential sorting methods

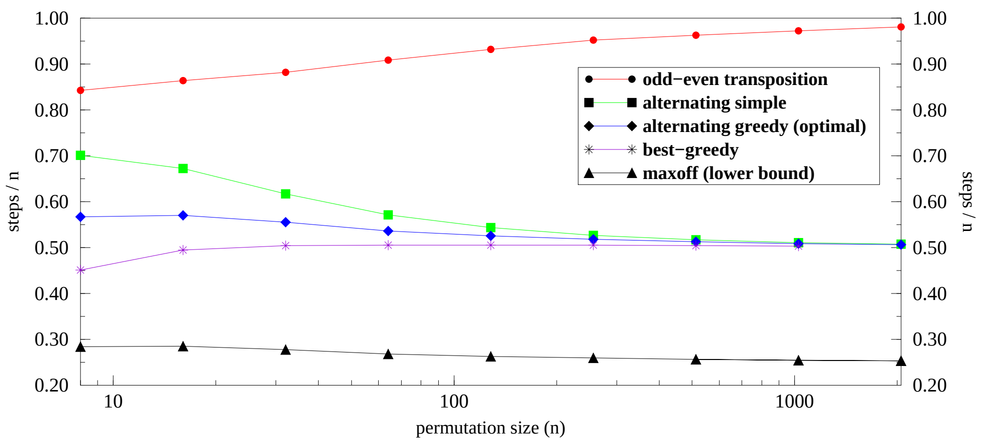

Figure 7: Average number of insertion steps divided by permutation size $n$. Only the odd-even transposition and alternating simple insertion sorts are efficiently implementable on a one-dimensional sub-bus array.

can be directly applied to our parallel sorting strategies.

## 6 Simulation Results

Other than our proof that alternating greedy sort is an optimal alternating sorting strategy, most of our results for two-way sorting strategies are lower bounds. Ultimately, we would like to say that the alternating simple strategy sorts nearly as well as an optimal alternating sorting strategy on average. To do so, we devised experiments to measure the average case performance of our sorting strategies. For several values of $n$ we generated a random sample of 1000 permutations of size $n$, and simulated the action of our sorting strategies for each permutation in the sample. The strategies simulated were early-stopping odd-even transposition sort, alternating simple insertion sort, alternating greedy sort, and best-greedy sort. In addition, to compare these with our lower bound, we computed the average value of maxoff($\pi$) over our sample permutations. Figure 7 shows the results, giving the average number of insertion steps taken divided by the permutation size.

Early-stopping odd-even transposition was the only one-way sorting strategy among the strategies simulated. As our lower bound of theorem 3.3 predicted, it sorts in nearly $n$ steps for large $n$.

Since alternating greedy is an optimal alternating sorting strategy, its graph equals twice the graph of maxoff($\pi$). This suggests that the lower bound of theorem 4.5 is nearly tight, since the graph converges to about 1/2. This also suggests that optimal alternating strategies use about half as many insertion steps as optimal one-way strategies for large $n$.

Unfortunately, the graph gives proof that alternating simple insertion sort is not optimal since its graph lies above alternating greedy sort and each strategy sorted the same set of permutations. A concrete example of a permutation for which the alternating simple insertion sorting strategy does not sort in the optimal number of steps is the permutation 4,3,2,1. Our simulations show that alternating simple insertion sort is nearly an optimal alternating strategy, especially for large $n$, since its graph also converges to about 1/2. Thus, alternating simple insertion sort takes nearly half the number of steps as odd-even transposition sort for large values of $n$.

In our simulations, we found a best-greedy sorting strategy for each permutation in the sample. The average number of best-greedy sorting steps converged to about $n/2$, showing that a best-greedy sort is not significantly better than optimal alternating sorts for large values of $n$.

Finally, the figure shows the sample mean of maxoff($\pi$) for our random permutation samples. Its graph converges to about 1/4. The large gap between this graph and the sorting strategy results suggests that maxoff($\pi$) may not be a good metric for determining the performance of general sorting strategies and that the lower bound in theorem 4.3 can likely be improved.

## 7 Implementing Sorting Strategies

In this section we give more detail on how one-dimensional sorting strategies can be implemented on a real sub-bus mesh computer such as a MasPar MP-1 or MP-2. In the process we will empirically compare the well-understood odd-even transposition sort and the simple insertion sorting strategies.

The sub-bus machine model we have adopted for this discussion consists of $n$ processors numbered 1 to $n$ which are linked together by a single segmentable communication bus. In this model, the sub-bus does not have wrap-around to directly connect processors 1 and $n$, whereas the MasPar sub-bus does have wrap-around. Like the MasPar, we adopt the single instruction multiple data (SIMD) computing model. Thus, there is a front-end processor which synchronously broadcasts parallel instructions to the processors. In addition, the front-end executes any sequential instructions in the program. In our programs there are two kinds of variables, *singular* which have a single value stored at the front-end and *plural* which have a value for each processor. A typical parallel instruction has the form:

**if** *test* **then** *statement.*

If the *test* is *True* at a processor, then the processor is said to be *active* and executes the *statement*. If the *test* is *False* at a processor, then the processor is said to be *inactive* and does not execute the *statement*. Statements to be executed include the usual kinds of RAM instructions such as addition, multiplication, and comparison. In our programming model we also have the communication instructions *left_shift*, *right_shift*, *left_broadcast* and *right_broadcast*. The shift operations perform nearest neighbor communication. For example, if every processor needs to get the value of $x$ from the processor to its immediate left and store it in $y$, all processors would execute:

$y \leftarrow$ *right_shift(x);*

Table 1: Effect of a right shift operation on a sub-bus machine. The * indicates that the value of $y$ does not change with the operation.

| | $y \leftarrow$ *right_shift*$(x)$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *proc-id* | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| $x$ | a | b | c | d | e | f | g | h | i |
| $y$ | * | a | b | c | d | e | f | g | h |

Here, $x$ and $y$ are plural variables. Table 1 illustrates the behavior of this statement. The more interesting communication operation is the sub-bus broadcast. For example, if a processor wants to broadcast the value of $x$ right on a sub-bus when a certain *test* is *True*, it would execute:

$$\textbf{if } \textit{test} \textbf{ then } y \leftarrow \textit{right\_broadcast}(x).$$

Each processor whose *test* is *True* places its value of $x$ on the sub-bus. The broadcast value travels to all the processors right of a broadcasting processor up to and including the next active processor. If there is no active processor to the right, then the broadcast travels to the end of the array. Thus, *all* processors (active and inactive) will read the value of $x$ from the first active processor to its left into its own variable $y$. If a processor has no active processor to its left, then the value of $y$ does not change. A *left_broadcast* behaves similarly, with data movement in the left direction. Table 2 illustrates the behavior of *right_broadcast*. Note that if *test* were true on every processor, a *right_broadcast*$(x)$ would be equivalent to a *right_shift*$(x)$. We differentiate them here because the shift operation is typically less expensive than a sub-bus broadcast. This is the case on the MasPar.

An additional primitive *singular* is needed for individual processors to communicate with the front-end. If the variable $x$ is plural, then *singular*$(i,x)$ returns a singular value which is the value of the plural variable $x$ at processor $i$. In our examples below we assume that all processors have "hard-wired" plural variables *proc_id*, the index of the processor, and *num_procs*, the number $n$ of processors.

We begin with the sub-bus pseudo-code to compute the OR of a plural boolean $x$ across all the processors in the array, which we'll refer to as *global_or*$(x)$. The *global_or* can be computed in a

Table 2: Effect of a right broadcast operation on a sub-bus machine. The * indicates that the value of $y$ does not change with the broadcast.

| | **if** *test* **then** $y \leftarrow$ *right_broadcast*$(x)$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *proc-id* | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| *test* | F | T | F | T | T | F | F | T | F |
| $x$ | a | b | c | d | e | f | g | h | i |
| $y$ | * | * | b | b | d | e | e | e | h |

constant number of sub-bus operations, and is useful in illustrating a typical sub-bus computation. The sub-bus pseudocode for *global_or* is given below.

```
singular global_or(plural x)
plural right_or;
right_or ← False;
if x then right_or ← left_broadcast(True);
right_or ← right_or or x;
return singular(1,right_or);
```

The plural variable *right_or* at each processor will eventually hold the OR of the values of $x$ at and to the right of the processor. If a processor has a *True* value for $x$ then it broadcasts a value of *True* to the left. At the end of the routine, the only way that *right_or* could have a value of *False* is if its value of $x$ was *False* and no processors to the right were active during the broadcast. To get the OR of $x$ over the entire array, we return the value of *right_or* at the leftmost processor. On many machines including the MasPar, there is actually significantly faster separate hardware support for computing the *global_or*.

To illustrate how sorting algorithms might be implemented on a sub-bus machine, we begin with the pseudo-code for odd-even transposition sort on the left half of figure 8. Normally, odd-even transposition sort has a loop that is executed exactly *num_procs* times. On the sub-bus you can determine whether the array is sorted with few sub-bus operations to perform early-stopping odd-even transposition sort. On even numbered steps even numbered processors look to their left and odd numbered processors look to their right in order to coordinate an interchange of out of order values. On odd numbered steps the reverse happens. At each iteration the processors check to see if the array is sorted. A total of two shift operations and one *global_or* are used to implement each step of odd-even transposition sort.

We can also implement simple insertion sorting steps in a constant number of sub-bus operations. As an example, consider the pseudocode of the left simple insertion sort on the right half of figure 8. The variable names in the program match fairly closely the description of the left simple insertion sorting strategy described in section 2.2. The set of pre-active processors are those which compute the value of *pre_active* equal to *True* on line (A). The blocking processors are those which have $pi < data$ in the calculation of *active* on line (C). The active processors are those that compute *active* equal to *True* on line (C). To complete the insertion step, a processor which is pre-active or is not active and does not have an active left neighbor accepts the value of its left neighbor which is stored in *left*, and a processor which is not active and whose left neighbor is active accepts the insertion stored in *data*.

An interesting thing to note is how we implicitly handle the boundary conditions. On line (B) we initialize *data* to $\infty$. If a processor has no processor to its right with *pre_active* equal to *True*, its value for *data* will remain set to $\infty$. This gives the same behavior as having an extra processor *num_procs + 1* with $data = \infty$ and *pre_active* = *True*. Similarly, on line (D) we set *left_active* to *True*, which is like having a processor 0 that has *active* equal to *True*. Thus, the implementation follows the abstract definition of left simple insertion steps given in section 2.2.

```
Odd_Even_Transposition_Sort(plural pi)
singular step, all_done;
plural left, right, done;
step ← 0;
all_done ← False;
while not all_done do
    step ← step + 1;
    left ← right_shift(pi);
    if proc_id = 1 then left ← −∞;
    right ← left_shift(pi);
    if proc_id = num_procs then right ← ∞;
    done ← pi < right;
    if proc_id ≡ step mod 2 then
        if left > pi then pi ← left;
    else
        if right < pi then pi ← right;
    endif
    all_done ← not global_or(not done);
endwhile
```

```
Left_Simple_Insertion_Sort(plural pi)
singular all_done;
plural left, data, pre_active, active, left_active;
all_done ← False;
while not all_done do
    left ← right_shift(pi);
    if proc_id = 1 then left ← −∞;
    pre_active ← left > pi;                                  (A)
    data ← ∞;                                                (B)
    if pre_active then
        data ← left_broadcast(pi);
    active ← pi < data or pre_active;                        (C)
    left_active ← True;                                      (D)
    left_active ← right_shift(active);
    if (not active or pre_active) and not left_active
      then pi ← left;
    if not active and left_active then pi ← data;
    all_done ← singular(1,data) = ∞;
endwhile
```

Figure 8: The sub-bus code to implement the odd-even transposition sorting strategy and the left simple insertion sorting strategy. The alternating simple sorting strategy has code complexity similar to the left simple insertion sorting strategy.

Termination is achieved by initializing the plural variable *data* to $\infty$ with each step. If processor 1's *data* remains $\infty$ after the pre-active processors broadcast, then the array must be sorted because there were no pre-active processors.

Note that right simple insertion steps can be implemented in a similar fashion, so we can easily extend this code to implement alternating simple insertion sort. Simple insertion steps can be implemented with two shift operations and one sub-bus broadcast. Simple insertion strategies can be implemented in roughly the same length and code complexity as odd-even transposition sort. Both sorts use two insertion steps per step, but alternating simple insertion step uses a sub-bus broadcast while odd-even transposition uses a *global_or*. The potential advantage of alternating simple insertion sort over odd-even transposition sort is that on average, the former sorts in half as many iterations than the latter.

Consider instead the problem of implementing an arbitrary parallel insertion sorting strategy. We can generalize the methods used by the previous two algorithms to provide a template for implementing any sorting strategy, given by the code *General_Insertion_Sort* in figure 9. Here, *Direction?* determines whether the insertion step for the strategy is a left or a right step and *Active?* determines whether a processor is active. Thus, the data movement involved in a insertion step can be implemented in a constant number of sub-bus operations. However, since the parallel insertion model made no restriction on the determination of the insertion step direction and the active set of

```
$General\_Insertion\_Sort$(plural $pi$)
singular $all\_done$;
plural $left, data, active, left\_active, right\_active$;
$all\_done \leftarrow False$;
while not $all\_done$ do
    if ($Direction?(\ldots) = Left$) then
        $active \leftarrow Active?(\ldots)$;
        $data \leftarrow \infty$
        if $active$ then $data \leftarrow left\_broadcast(pi)$;
        $left\_active \leftarrow True$;
        $left\_active \leftarrow right\_shift(active)$;
        if $data \neq \infty$ then
            if not $left\_active$ then $pi \leftarrow left$;
            if not $active$ and $left\_active$ then $pi \leftarrow data$;
        endif
    else
        ...right step...
    endif
    $left \leftarrow right\_shift(pi)$;
    $all\_done \leftarrow$ not $global\_or(left > pi)$;
endwhile
```

Figure 9: The sub-bus code to implement the data movement in an arbitrary sorting strategy. The functions *Direction?* and *Active?* determine the insertion step direction and whether a processor is active, respectively, for that sorting strategy.

processors, the computation of *Direction?* and *Active?* could be arbitrarily complex. For example, if we were implementing the left greedy strategy, the code for *Active?* would consist of computing the minimum value of *pi* of the processors to the right and comparing that with the value of *pi* at the processor. This suffix minimum computation requires $\Theta(\log n)$ sub-bus operations [2]. A logarithmic number of sub-bus steps per insertion step implies that the greedy strategies are unsuitable for practical use on a sub-bus machine. Thus, only the odd-even transposition sorting strategy and the sorting strategies based on simple insertion steps are efficiently implementable among the strategies we have discussed.

## 8 Implementation Results

Our simulation results indicate that alternating simple insertion sort takes about half as many insertion steps as odd-even transposition sort. To see if this improvement factor holds in practice we implemented the shearsort two-dimensional mesh sorting algorithm using some of the one-dimensional sorts we have described.

Table 3: Sub-bus operations per insertion step of the shearsort algorithms.

| | *shift* | *broadcast* | *global_or* |
|---|---|---|---|
| Oblivious Odd-Even Shearsort | 2 | 0 | 0 |
| Odd-Even Shearsort | 2 | 0 | 1 |
| Simple Shearsort | 2 | 1 | 1 |

The shearsort algorithm [16] on an $n \times n$ array of processors proceeds in $2\lceil \log_2 n \rceil + 1$ phases. The even phases sort the columns of the array independently, moving the smallest values to the top. The odd phases sort the rows of the array, with ordering values in the odd rows from smallest on the left to largest on the right and ordering values in even rows from largest on the left to smallest on the right. The row and column sorts are done using a one-dimensional sort. The result is that the elements are sorted in "snake order."

We implemented three versions of the shearsort algorithm. The first version of shearsort is *simple shearsort* which uses alternating simple insertion sort as the one-dimensional algorithm. The second version of shearsort is *odd-even shearsort* which uses the early-stopping version of odd-even transposition as the one-dimensional algorithm. The third version of shearsort is *oblivious odd-even shearsort* which employs the oblivious odd-even transposition sort, using $n$ insertion steps per one-dimensional sort.

The implementation was done in the MPL programming language that is similar to C, but has constructs to support the programming style given by our sub-bus pseudocode earlier. The programs were executed on a square 16,384 ($128 \times 128$) processor MasPar MP-1. Each program was executed on the same set of randomly chosen permutations on $n \times n$ arrays of processors where $n$ ranged between 8 to 128.

In our programs we used the MPL equivalents of the *shift*, *broadcast*, and *global_or* communication primitives. An important difference between the programs is the number and kinds of operations needed to execute an insertion step. Odd-even shearsort uses early-stopping odd-even transposition sorting steps. Recall that our implementation of early-stopping odd-even transposition sorting used two *shift* operations to perform the insertion step and one *global_or* to check if the array is sorted. A phase of odd-even shearsort ends when all the one-dimensional sorts have completed. This means that we must check if the entire two-dimensional array is sorted with each sorting step, and so the *global_or* to determine termination now acts over the entire array. Simple shearsort uses simple insertion steps which consisted of two *shift* operations and one *broadcast*. An additional *global_or* was added to determine whether all the one-dimensional sorts had completed. Since oblivious odd-even shearsort does not check if the array is sorted, it uses only two shift operations per insertion step. Table 3 summarizes the sub-bus operations needed per insertion step of each algorithm.

The differing operation counts results in different insertion step costs for each algorithm. Figure 10 shows the average time per insertion step for each shearsort implementation as given by our experiments. The early-stopping mechanism of odd-even insertion steps incurs a 20% overhead over oblivious odd-even insertion steps. However, this overhead is small relative to the simple shearsort: simple insertion steps are nearly twice as costly as both odd-even transposition insertion steps. In addition, simple shearsort shows a slight rise in time per step because as $n$ grows so does the cost

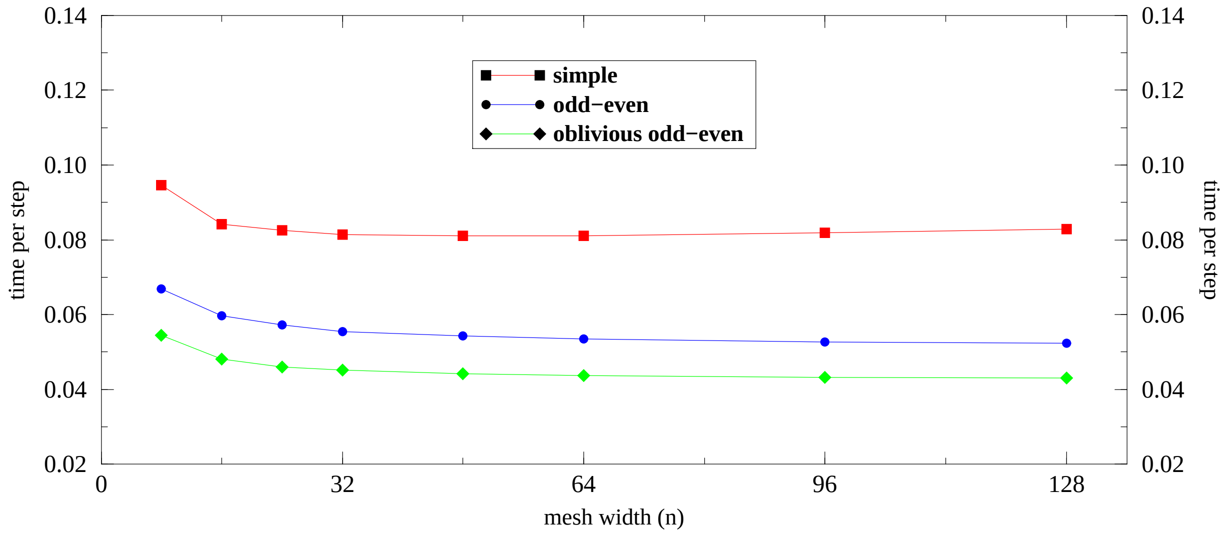

Figure 10: Time per insertion step in milliseconds for each Shearsort.

of the broadcast operation.

Figure 11 shows the average number of insertion steps used by the three shearsorts divided by the mesh width $n$. One might expect these results to mimic our simulation results of section 6 with an additional $2\lceil\log_2 n\rceil + 1$ factor due to the number of shearsort phases. Indeed, simple shearsort uses nearly half as many insertion steps as oblivious odd-even shearsort for mesh widths of sixteen processors or larger. What is somewhat surprising is that odd-even shearsort uses nearly as few insertion steps as simple shearsort, using only about 10% more. Even though the two-dimensional inputs to the shearsort algorithms were chosen uniformly at random, the inputs to the one-dimensional sorting subroutines are not. As the two-dimensional data is sorted, the one-dimensional sorting phases may need to perform less work. One possible explanation for the low number of insertion steps used by odd-even shearsort is that the sorting work needed in the later phases of the algorithm may require only local rearrangements. If this is the case, then odd-even transposition steps are as effective as simple insertion steps.

Figure 12 shows the time (divided by $n$) for each of our shearsorts. Odd-even shearsort has the best performance of our shearsort algorithms. It uses nearly as few insertion steps as simple shearsort (relative to oblivious odd-even shearsort) at a cost per step competitive with oblivious odd-even shearsort (relative to simple shearsort). The factor of two improvement in the number of insertion steps gained by simple shearsort over oblivious odd-even shearsort is offset by the factor of two expense of each insertion step. This results in oblivious odd-even shearsort and simple shearsort having similar performance for large $n$. Odd-even shearsort performs 30% faster than simple and oblivious odd-even shearsort for large $n$.

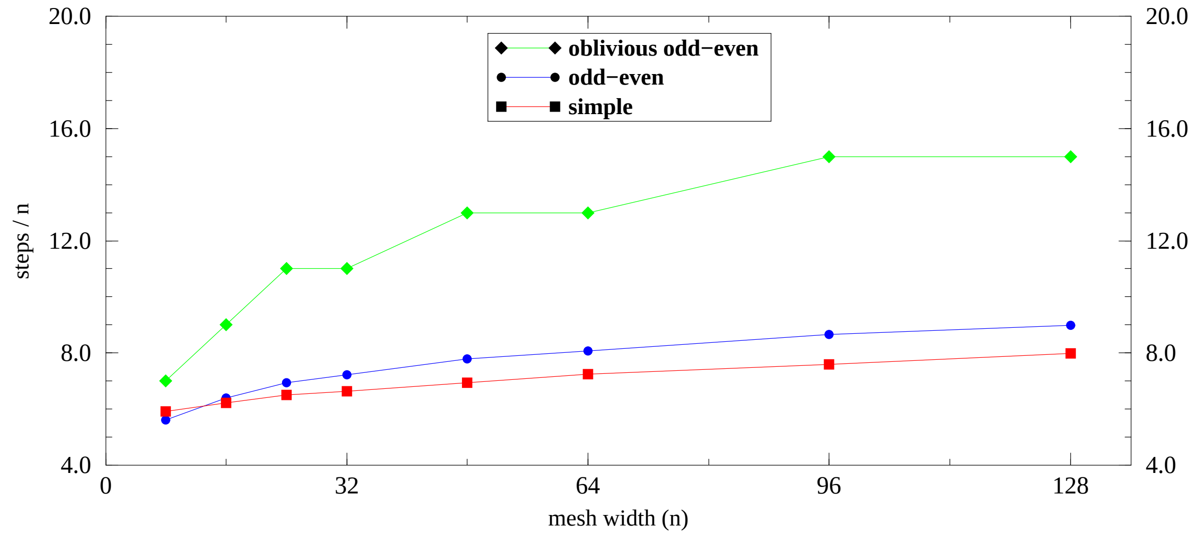

Figure 11: Number of sorting steps divided by mesh width for each Shearsort.

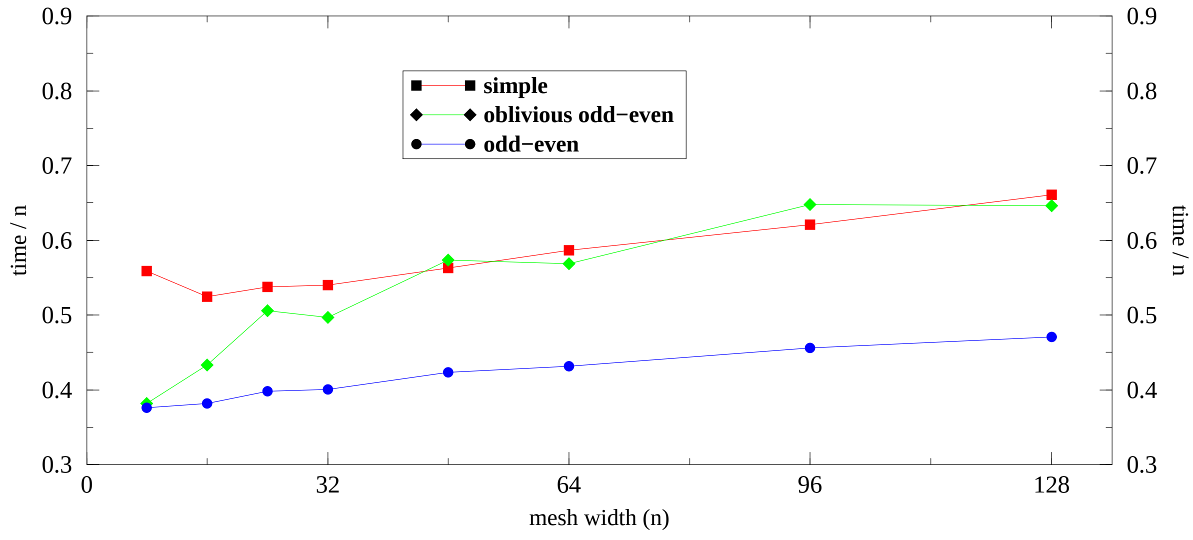

Figure 12: Sorting time in milliseconds divided by mesh width for each Shearsort.

## 9 Conclusion

In this paper we have investigated one-dimensional sorting algorithms which take advantage of the sub-bus architecture of some parallel computers. The parallel insertion model provides a framework for investigating these algorithms in the form of one-way, alternating, and two-way sorting strategies. In the parallel insertion model, the left greedy sorting strategy and the left simple insertion sorting strategy are optimal one-way sorting strategies, whereas the odd-even transposition sorting strategy is not. The alternating greedy sorting strategy is an optimal alternating sorting strategy. From our simulation study, a best-greedy strategy is only slightly better than an optimal alternating sorting strategy and the alternating simple insertion sorting strategy is only slightly worse than an optimal alternating strategy. A proof of these two observations is still unknown. All our two-way strategies take about half the number of insertion steps as any one-way sorting strategy and odd-even transposition sort.

A remaining question in the parallel insertion model is whether a general sorting strategy can be found that performs significantly better than optimal alternating strategies. Our hunch is that this is not the case, and that a better lower bound can be found.

From our empirical study, shearsort using alternating simple insertion sort is no faster than shearsort using oblivious odd-even transposition sort, even though it uses half as many steps in the parallel insertion model. The fastest version of shearsort in our experiments used the early-stopping version of odd-even transposition sort. It used nearly as few insertion steps as simple shearsort and its insertion steps were less costly. These results demonstrate that one must carefully consider the costs of insertion steps when implementing any sorting strategy on a real sub-bus machine.

We would not want to leave the reader with the impression that shearsort using the early-stopping odd-even transposition sort is the best sorting algorithm for the MasPar MP-1. The MasPar not only has the sub-bus mesh which we have been investigating, but it also has a separate communication architecture called the *router*. Sorting using the router is much faster than sorting using the mesh.

## Acknowledgements

Thanks to Peter VanVleet for helping us use and understand the MasPar and MPL. Thanks also to Purdue University for use of its 16,384 processor MasPar MP-1 in this project.

## References

[1] T. Blank. The MasPar MP-1 architecture. In *Proceedings of COMPCON Spring 90 - The Thirty-Fifth IEEE Computer Society International Conference*, pp. 20–24, February 1990.

[2] A. Condon, R. E. Ladner, J. Lampe, and R. Sinha. Complexity of sub-bus mesh computations. *SIAM Journal on Computing*, Vol. 25, No. 3, pp. 520–539, 1996.

[3] M. H. DeGroot. *Probability and Statistics*, Addison-Wesley, 1975, 1986.

[4] H. B. Demuth. *Electronic Data Sorting.* PhD thesis, Stanford University, October, 1956.

[5] H. B. Demuth, Electronic Data Sorting. *IEEE Transactions on Computers*, Vol. 34, No. 4, pp. 296–310, 1985.

[6] J. D. Fix and R. E. Ladner. Optimal one-way sorting on a one-dimensional sub-bus array. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 586–594, January 1995.

[7] N. Haberman. Parallel neighbor-sort (or the glory of the induction principle). Technical Report AD-759 248, National Technical Information Service, 1972.

[8] D. E. Knuth. *The Art of Computer Programming Volume I: Fundamental Algorithms.* Addison-Wesley, 1973, 1968.

[9] D. E. Knuth. *The Art of Computer Programming Volume III: Sorting and Searching.* Addison-Wesley, 1973, 1968.

[10] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

[11] J. Y-T. Leung and S. M. Shende. On Multi-dimensional Packet Routing for Meshes with Buses. *Journal of Parallel and Distributed Computing*, Vol. 20, No. 2, pp. 187–197, 1994.

[12] R. Miller, V. K. Prasanna-Kumar, D. I. Reisis, and Q. F. Stout. Parallel Computations on Reconfigurable Meshes. *IEEE Transactions on Computers*, Vol. 42, No. 6, pp. 678–692, 1993.

[13] V. K. Prasanna-Kumar and C. S. Raghavendra. Array Processor with Multiple Broadcasting. *Journal of Parallel and Distributed Computing* Vol. 4, No. 2, pp. 173–190, 1987.

[14] S. Rajasekaran. Mesh Connected Computers with Fixed and Reconfigurable Buses: Packet Routing and Sorting. *IEEE Transactions on Computers* Vol. 45, No. 5, pp. 529–539, 1996.

[15] K. Sado and Y. Igarashi. Some parallel sorts on a mesh-connected processor array and their time efficiency. *Journal of Parallel and Distributed Computing*, Vol. 3, pp. 398–410, 1986.

[16] I. Scherson, S. Sen, and A. Shamir. Shear-sort: A true two-dimensional sorting technique for VLSI networks. In *IEEE-ACM International Conference on Parallel Processing*, pp. 903–908, 1986.

[17] C. Schnorr and A. Shamir. An optimal sorting algorithm for mesh connected computers. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pp. 255–263, 1986.