Optimal One-Way Sorting on a One-Dimensional Sub-Bus Array*

James D. Fix† Richard E. Ladner‡

Abstract

The problem of sorting on a one-dimensional sub-bus array of processors is addressed. 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 is implemented on the MasPar MP-1 and MP-2 parallel computers. The sub-bus broadcast operation makes possible a new class of parallel sorting algorithms which can be characterized by *parallel insertion steps*. We restrict our attention to the class of sorting methods where parallel insertion steps are in the same direction, say left. For this class of *one-way sorting strategies* we prove a lower bound and give two strategies, the *left greedy sort* and the *left adaptive insertion sort*, both of which achieve the lower bound. Because our parallel insertion model is quite general, it is not necessarily the case that a sorting strategy in the model can be efficiently implemented on a real sub-bus array of processors. The left greedy sort cannot be efficiently implemented, but the left adaptive insertion sort can be efficiently implemented. The two sorting strategies have different properties and each is interesting in its own right.

1 Introduction

Sorting is a fundamental problem in computation. It has been well studied in many forms (see Knuth [4], for example) and occurs as a sub-problem of a variety of algorithms and applications. For many applications, sorting problem sizes are large enough to warrant the need for efficient sorting algorithms for massively parallel machines. A popular architecture for parallel computers is the two-dimensional array or mesh of processors such as the MasPar MP-1 and MP-2 [1], and the Intel Paragon. The problem of sorting on a mesh of processors has been well studied [5, 6, 7]. However, most of the sorting techniques for meshes assume that a processor can only communicate with its nearest neighbors. However, modern mesh computers, such as the MasPar MP-1 and MP-2 allow for broadcast along rows and columns. Moreover, any bus connecting a row or column can be broken up into sub-buses allowing so-called segmented broadcasts. This paper addresses the question of whether or not the sub-bus capability gives rise to better sorting algorithms than are possible on the standard mesh of processors which only allows nearest neighbor communication.

A basic building block of most two-dimensional mesh sorting algorithms is the one-dimensional algorithm called *odd-even transposition sort* described in Leighton's book [5]. Odd-even transposition sort resembles a parallel "bubble sort". It involves successively comparing and exchanging values between odd/even neighbors and even/odd neighbors until the values are in order. Odd-even transposition sorts any input in exactly d or $d+1$ compare-exchange steps, where d is the maximum distance a value must travel to its destination. This is a tight bound since d mesh communication steps are necessary to transmit this value to its final destination. This bound no longer holds in the one-dimensional sub-bus model as can be seen by the following example. Consider the data $2, 3, ..., n, 1$ stored in processors $1, 2, 3, ..., n$, respectively. The standard odd-even transposition sort takes $n-1$ or n steps to sort the data. However, a single left broadcast by processor n to all the other processors can inform each one to the left to shift its data one processor to the right and inform processor 1 to accept the data from processor n. Thus, sorting this data array can be done in constant time on a one-dimensional sub-bus array of processors.

2 Summary of Results

In this paper we focus on one-dimensional sub-bus sorting algorithms in which broadcasts of a distance greater than one are in just one direction, say left. We call these algorithms *one-way sorting strategies*. We begin by introducing a simple abstract model called the parallel insertion model to model one-dimensional, in-place, comparison based, sorting strategies[1]. A left insertion step in the parallel insertion model resembles a set of simultaneous insertion sort steps. A left-only

*This paper appears in the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, January 1995. It is available as a University of Washington, Department of Computer Science and Engineering Technical Report, TR 94-11-01.

†Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, fix@cs.washington.edu. Fix's research was supported by an Osberg Family Trust Fellowship.

‡Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, ladner@cs.washington.edu. Ladner's research was supported by NSF grant no. CCR-9108314.

[1] A strategy is more general than an algorithm. A strategy to sort a data array is simply a sequence of allowable steps which sorts the array. The sequence may or may not be described by an algorithm.

sorting strategy is one that only uses left insertion steps.

We show that that the number of steps required to sort using a left-only sorting strategy 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 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 introduce the left greedy sorting strategy which always sorts in the minimum number of left insertion steps. The left greedy sorting strategy has the interesting property that each step in the strategy removes the maximum number of inversions possible by any left insertion step. Unfortunately, the left greedy sorting strategy is not efficiently implementable on a real sub-bus array.

Finally, we introduce the left adaptive insertion sorting strategy which is also optimal, but can be implemented efficiently on a real sub-bus array. In the context of the parallel insertion model, the left adaptive insertion sort meets our goal of finding an algorithm which is as good as odd-even transposition sort on all inputs and outperforms it on some inputs. Our lower bound on the expected number of steps for left-only sorting strategies indicates that the advantage of left adaptive insertion sort over odd-even transposition sort is quite small on average.

In summary, we introduce a new realistic model for sorting. For an interesting subclass of sorting strategies in the model, the one-way sorting strategies, we demonstrate lower bounds and prove the optimality of several interesting sorting strategies. We briefly describe some preliminary results on two-way sorting strategies.

3 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 π of $\{1, 2, ..., n\}$ where n is the number of processors. The data value $\pi[i]$ is stored at processor i and π is sorted if $\pi[i] = i$ for $1 \leq i \leq n$. A *left insertion step* β is defined by a set of *active* processors $A_\beta \subseteq \{0, 1, 2, ..., n\}$, where 0 is always a member of A_β[2]. The permutation $\beta(\pi)$ is defined to be 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

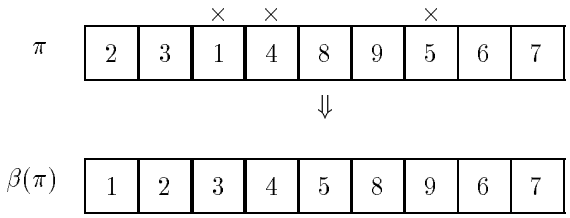

Figure 1: Data movement in a left insertion step β applied to π where $A_\beta = \{0, 3, 4, 7\}$

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 β applied to π is the permutation $\beta(\pi)$ where

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

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.

A *left-only sorting strategy for* π is a sequence $\beta_1, \beta_2, ...\beta_T$ where $\pi = \pi_0$, for $1 \leq i \leq T$, $\pi_i = \beta_i(\pi_{i-1})$, and π_T is sorted. The left-only sorting strategy $\beta_1, \beta_2, ...\beta_T$ sorts π in T steps.

In a symmetric way we can define *right insertion steps* and *right-only sorting strategies*. A more general sorting strategy is a *two-way sorting strategy* which allows both left and right insertion steps. In this paper we focus on *left-only sorting strategies* because there are interesting examples of such sorting strategies which can be shown to be optimal. In section 9 we return to two-way sorting strategies, giving some preliminary empirical and theoretical results.

4 Lower Bounds

Left-only sorting strategies are limited by the fact that an 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 π be a permutation of $\{1, 2, \ldots, n\}$. Suppose i is the location of the value x in π, so $x = \pi[i]$. Then we define $\text{dist}_L(x, \pi) = \max\{0, x - i\}$. Since the final destination of x is processor x, $\text{dist}_L(x, \pi)$ is the distance in π of x from it final destination if x is to the left of its final destination. Otherwise, $\text{dist}_L(x, \pi) = 0$. Define $\text{maxdist}_L(\pi) = \max_x \text{dist}_L(x, \pi)$, that is $\text{maxdist}_L(\pi)$ is the maximum distance left to its final destination of any value in π.

We observe that π is sorted if and only if $\text{maxdist}_L(\pi) = 0$. Clearly, if π is sorted then

[2] We add 0 as a dummy active processor and assume $\pi[0] = -\infty$ This makes the definitions and proofs go through more smoothly.

$\text{maxdist}_L(\pi) = 0$. If π 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,

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

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

THEOREM 4.1. *If $\beta_1, \beta_2, \ldots \beta_T$ is a left-only sorting strategy for π, then $T \geq maxdist_L(\pi)$.*

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

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

THEOREM 4.2. *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 π 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 π 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 $n \geq k + 1$. We calculate:

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

□

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 [3]). Combining this fact with the previous two theorems we have:

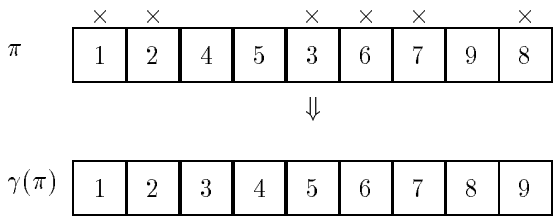

Figure 2: Data movement in a left greedy step.

THEOREM 4.3. *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 odd-even transposition sort can only be slight. Nonetheless, it is interesting to see if optimality for left-only sorting strategies can be achieved.

5 The Left Greedy Sorting Strategy

A natural approach for devising sorting strategies on the sub-bus is to try to maximize the sorting work performed at each step. The *left greedy sorting strategy* was designed to maximize the number of inversions removed by a step. The *set of inversions* in a permutation π 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 [4]. The only permutation with no inversions is the sorted permutation. Define the *left greedy step for* π as a left insertion step where the set of active processors is $G_L(\pi) = \{i \mid \text{for all } j > i,\ \pi[j] > \pi[i]\} \cup \{0\}$. An example of the action of a left greedy 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 step is that it never creates any new inversions. This is stated in the following lemma.

LEMMA 5.1. *Let π be a permutation, γ be a left-greedy step for π. 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, $\pi[l] > \pi[i]$ for all $l > i$. As

a consequence $k \in G_L(\pi)$. This is impossible because $\gamma(\pi)[j] = \pi[i]$ implies $j > k$. □

Our next theorem demonstrates that a left greedy step truly exhibits greedy behavior.

THEOREM 5.1. *Let γ be the left greedy step for π. For all possible left insertion steps β for π, $|I(\gamma(\pi))| \leq |I(\beta(\pi))|$.*

Proof. Let ω be a left broadcast step with the property that for all left broadcast steps β, $|I(\omega(\pi))| \leq |I(\beta(\pi))|$ and among those with the property A_ω 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_ω. Define left insertion step ν by the active set $A_\nu = A_\omega \cup \{i+1, i+2, \ldots, n\}$. It follows that $\nu(\pi) = \omega(\pi)$ but A_ν has more more members than A_ω, which contradicts our choice of ω.

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 ν from ω 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 5.1 making i active in ν will not introduce any inversions than ω 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 ν from ω 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_ω 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 5.1, no new inversions are created by removing i from A_ω, that is, $I(\nu(\pi)) \subseteq I(\omega(\pi))$. On the other hand, the removal of i from A_ω 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 π is the left-only sorting strategy for π where each insertion step is left greedy. The left greedy sorting strategy is an optimal left-only sorting strategy as evidenced by the following theorem.

THEOREM 5.2. *Let π be a permutation and $\gamma_1, \gamma_2, \ldots \gamma_T$ be the left greedy sorting strategy for π. Then $T = maxdist_L(\pi)$.*

Proof. Let π be a permutation and $\gamma_1, \gamma_2, \ldots \gamma_T$ be the left greedy sorting strategy for π. By the lower bound theorem, to prove optimality we have to show $T \leq \text{maxdist}_L(\pi)$. To do this we simply have to show that each left adaptive insertion step reduces the maximum distance left of values in π by one. That is, for $1 \leq i \leq T$, $\text{maxdist}_L(\pi_i) = \text{maxdist}_L(\pi_{i-1}) - 1$, where $\pi_0 = \pi$ and $\pi_i = \gamma_i(\pi_{i-1})$. This follows from lemma 5.2 below. □

LEMMA 5.2. *Let π be an unsorted permutation and γ be the greedy step for π. 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 adaptive 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 $x = \pi[i]$.

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

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 broadcast steps, $\gamma(\pi)[i+1] = x$. If $\text{dist}_L(x, \pi) = 0$ then $x - i \leq 0$, so $x - i - 1 < 0$. Hence, $\text{dist}_L(x, \gamma(\pi)) = \max\{0, x-i-1\} = 0$. If $\text{dist}_L(x, \pi) > 0$ we have $x - i > 0$, so $x - i - 1 \geq 0$. It follows that $\text{dist}_L(x, \gamma(\pi)) = \max\{0, x - i - 1\} = \text{dist}_L(x, \pi) - 1$.

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

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 γ. By lemma 5.1, 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 π 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 $\text{dist}_L(x, \gamma(\pi)) = \max\{0, x - k\} = 0$.

Conclusion of the proof of lemma 5.2: Having established these claims, the lemma follows easily:

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

The first equality follows from the definition of

maxdist_L, the second equality from claim 2, and the last equality from claim 1. □

6 The Left Adaptive Insertion Sorting Strategy

From the standpoint of the parallel insertion model the left greedy sorting strategy is as good as any left-only sorting strategy. Unfortunately, it is not always the case that a sorting strategy for the parallel insertion model can be implemented efficiently on a real machine. In order to implement a sorting strategy efficiently there must be a way for processors to quickly determine if they are active or inactive. In the left greedy sorting strategy, each processor has to determine if its value is smaller than the value at all higher numbered processors. This task could be performed by a parallel suffix minimum operation, but such an operation requires $\log n$ steps on the one-dimensional sub-bus mesh [2]. A logarithmic number of sub-bus steps per left insertion step implies that the left-only greedy algorithm is unsuitable for practice.

We have discovered an optimal left-only sorting strategy, the *left adaptive insertion sorting strategy*, which is implementable in a constant number of sub-bus operations per left insertion step. The description of which processors are active in a left adaptive insertion step is fairly complicated, but a left adaptive insertion step can be implemented on a real sub-bus array in roughly the same complexity as a step in the odd-even transposition sort.

Determining the active processors in a left adaptive insertion step for π is a two step process. First, we define the *pre-active* set of processors $P_L(\pi)$ to be

$$P_L(\pi) = \{i \mid \pi[i-1] > \pi[i] \text{ and } \pi[i] < \pi[i+1]\}$$

where $1 \leq i \leq n$ and the boundary values $\pi[0] = -\infty$ and $\pi[n+1] = \infty$. 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 broadcasts of 3 and 8 are broadcast as far left as possible, creating inversions. To correct this, we extend the active set in the obvious way: given a permutation π, define the *blocking* set of processors

$$B_L(\pi) = \{i \mid \text{for some } j > i, j \in P_L(\pi), \pi[i] < \pi[j], \text{ and for } i \leq l < j, l \notin P_L(\pi)\}$$

The additional active processors simply insure that a broadcast value does not create any inversions. As we shall see in the proof of lemma 7.1, claim 3, the blocking processors within a segment bounded by two pre-active processors always form a contiguous block at the left

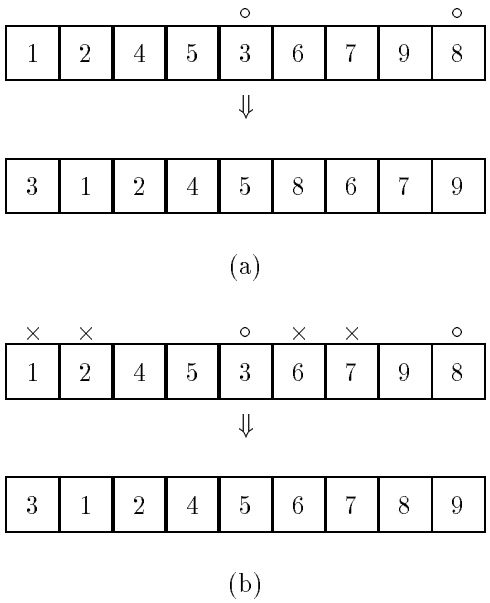

Figure 3: (a) Inversions created by left insertion step with active set $P_L(\pi)$. (b) Corrected left insertion step with additional active processors $B_L(\pi)$.

end of the segment. These blocking processors block the broadcast of larger values to their right. 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 adaptive insertion step for* π is a left insertion step whose set of active processors is the combined set $A_L(\pi) = P_L(\pi) \cup B_L(\pi) \cup \{0\}$. The *left adaptive insertion sorting strategy for* π is the left-only sorting strategy where each insertion step is a left adaptive insertion step. Figure 4 shows the left adaptive insertion for a sample permutation.

To see that left adaptive insertion sort is efficiently implementable, first note that a processor can determine if it is pre-active using a left and right nearest neighbor communication. For a processor to determine if it blocking, each pre-active processor broadcasts its value to the left. A processor is blocking if it is not pre-active and the value it observes from a pre-active processor is larger than its own.

7 Optimality of Left Adaptive Insertion Sort

We now prove the optimality of the left adaptive insertion sort as stated in the theorem below.

THEOREM 7.1. *Let π be a permutation and $\alpha_1, \alpha_2, \ldots \alpha_T$ be the left adaptive insertion sorting strategy for π. Then $T = \text{maxdist}_L(\pi)$.*

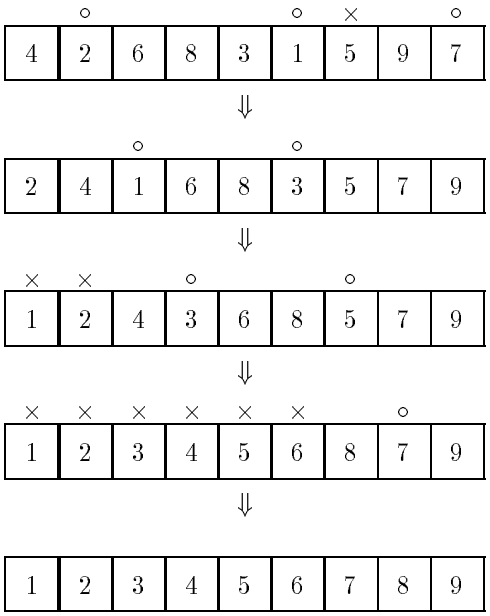

Figure 4: left adaptive insertion sort

Proof. Let π be a permutation and $\alpha_1, \alpha_2, \ldots \alpha_T$ be the left adaptive insertion sorting strategy for π. By the lower bound theorem, to prove optimality we have to show $T \leq \text{maxdist}_L(\pi)$. To do this we simply have to show that each left adaptive insertion step reduces the maximum distance left of values in π by one. That is, for $1 \leq i \leq T$, $\text{maxdist}_L(\pi_i) = \text{maxdist}_L(\pi_{i-1}) - 1$, where $\pi_0 = \pi$ and $\pi_i = \alpha_i(\pi_{i-1})$. This follows from lemma 7.1 below. □

LEMMA 7.1. *Let π be an unsorted permutation and α be the left adaptive insertion step for π. Then $maxdist_L(\alpha(\pi)) = maxdist_L(\pi) - 1$.*

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 adaptive insertion step. The second claim states that the value at a pre-active processor after the left adaptive 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 adaptive 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 maximim distance left of the values must decrease by one as a result of the left adaptive insertion step.

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

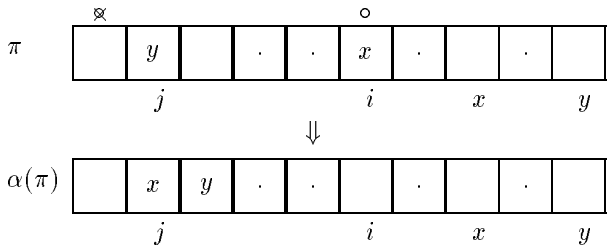

Figure 5: Data movement of x and y with left adaptive insertion step α described in Claim 2.

CLAIM 1: If $i \notin A_L(\pi)$ then $\text{dist}_L(x, \alpha(\pi)) = \text{dist}_L(x, \pi) = 0$ or $\text{dist}_L(x, \alpha(\pi)) = \text{dist}_L(x, \pi) - 1$.

Suppose $i \notin A_L(\pi)$. There are two cases: there is a $j \in A_L(\pi)$ with $j > i$ or for all $l > i$ we have $l \notin A_L(\pi)$.

Case 1. Let $j \in A_L(\pi)$ with $j > i$ and for $i < l < j$, $l \notin A_L(\pi)$. By the definition of left insertion steps we have $\alpha(\pi)[i+1] = x$. Thus, either $\text{dist}_L(x, \alpha(\pi)) = \text{dist}_L(x, \pi) = 0$ or $\text{dist}_L(x, \alpha(\pi)) = \text{dist}_L(x, \pi) - 1$.

Case 2. Suppose that for all $l > i$ we have $l \notin A_L(\pi)$. It must be the case that for $i \leq l < n$, $\pi[l] < \pi[l+1]$. Otherwise, there is a maximal l with $i \leq l < n$ and $\pi[l] > \pi[l+1]$. If $l + 1 = n$ we have $l \in P_L(\pi)$. If $l + 1 < n$ then $\pi[l+1] < \pi[l+2]$ by our choice of l, so $l \in P_L(\pi)$. But $l \notin P_L(\pi)$, so we have a contradiction. Hence, for $i \leq l < n$, $\pi[l] < \pi[l+1]$.

Since there are no active processors right of i, $\alpha(\pi)[i] = x$ and $\text{dist}_L(x, \pi) = \text{dist}_L(x, \alpha(\pi))$. There are at least $n - i$ values greater than x, so $x \leq n - (n - i) = i$. Thus, $\text{dist}_L(x, \alpha(\pi)) = 0$.

CLAIM 2: If $i \in P_L(\pi)$ then there is a $j \notin A_L(\pi)$ with $y = \pi[j]$ and $\text{dist}_L(x, \alpha(\pi)) \leq \text{dist}_L(y, \alpha(\pi))$.

Suppose $i \in P_L(\pi)$. Let $\alpha(\pi)[j] = x$. Since $i \in P_L(\pi)$, $\pi[i-1] > \pi[i]$. Hence $i - 1 \notin P_L(\pi)$. Thus, $j < i$ and $j \notin P_L(\pi)$.

Let $y = \pi[j]$. It must be true that $y > x$, otherwise $j \in B_L(\pi)$, so $y \geq x + 1$. Note that $\alpha(\pi)[j+1] = y$ so we have

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

Therefore, Claim 2 follows with this choice of $j \notin A_L(\pi)$.

Claim 2 is illustrated in Figure 5. For x to be farther left of its destination than y, x would have to be greater than y. But $x < y$ instead so x's distance left is bounded by the left distance of y.

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

This is the most difficult proof and requires an induction on contiguous sequences of blocking processors. Suppose $i \in B_L(\pi)$. By definition of $B_L(\pi)$ there is a $j \in P_L(\pi)$ such that $y = \pi[j] > x$ and for $i \leq l < j$, $l \notin P_L(\pi)$.

Let $\alpha(\pi)[k] = y$. We know that $k \neq j$, otherwise $j-1 \in A_L(\pi)$. This cannot be since this would imply that $\pi[j-1] < \pi[j]$ and $j \notin P_L(\pi)$.

Thus, for y to move to processor $k < j$ with step α, we must have $k \notin A_L(\pi)$ and $k-1 \in A_L(\pi)$. In fact, $k-1 \in B_L(\pi)$ by the definition of j. We show by induction on $k-l$ that if $i \leq l \leq k-1$, then $l \in B_L(\pi)$ and $\pi[l] < \pi[l+1]$:

Basis: $l = k-1$. From above we know that $k-1 \in B_L(\pi)$. Thus, $\pi[k-1] < y$. Since $k \notin B_L(\pi)$, we know that $\pi[k] > y$, so $\pi[k-1] < \pi[k]$.

Inductive Step: Assume that $l \in B_L(\pi)$, $\pi[l] < \pi[l+1]$ and that $i < l \leq k-1$. We know that $\pi[l-1] < \pi[l]$, otherwise $l \in P_L(\pi)$. By the definition of $B_L(\pi)$, $\pi[l] < y$, so $\pi[l-1] < y$. We know that $l-1 \notin P_L(\pi)$ by definition of j, so it must be true that $l-1 \in B_L(\pi)$.

So for all l with $i \leq l < k$, we have $\pi[l] < \pi[l+1]$. Since $\pi[k-1] < y$ we can say that $x = \pi[i] \leq y-(k-i)$. Also, we know that $x = \pi[i] < \pi[i+1]$. So for $i > 1$ it must be true that $\pi[i-1] < \pi[i] = x$, otherwise $i \in P_L(\pi)$. It follows that $i-1 \in A_L(\pi)$. Therefore, whether $i = 1$ or $i > 1$, we have $\alpha(\pi)[i] = x$.

We can now finish the proof of the final case:

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

The inequality step above follows because $x+k-i \leq y$. So there exists a $j \in P_L(\pi)$ with $y = \pi[j]$ whose distance left is at least as large as x's in $\alpha(\pi)$.

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. This extra distance left is balanced by the fact that $x \leq y-(k-i)$, giving a bound on $\mathrm{dist}_L(x, \alpha(\pi))$.

Conclusion of the proof of lemma 7.1: Having established these three claims, the lemma follows easily:

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

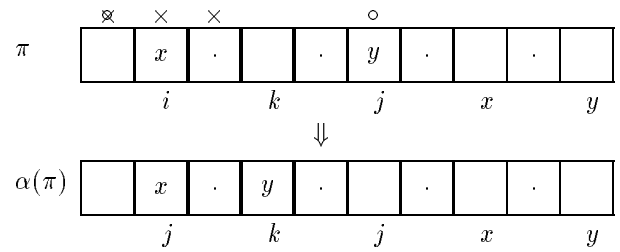

Figure 6: Data movement of x and y with left adaptive insertion step α described in Claim 3.

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 adaptive insertion step. □

8 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 compare the well understood odd-even transposition sort and the left adaptive insertion sort.

There are n processors numbered 1 to n which are linked together by a single segmentable communication bus. For simplicity 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_broadcast* and *right_broadcast.* For example, if a processor wants to broadcast a value x right on a sub-bus when a certain *test* is *True*, it would execute:

if *test* **then** $y \leftarrow$ *right_broadcast(x).*

Each processor whose *test* is *True* places its value

	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

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

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 register 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 1 illustrates the behavior of *right_broadcast*. An additional primitive *sing* is needed for individual processors to communicate with the front-end. If the variable x is plural, then *sing(i,x)* returns a singular value which is the value of 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 program for odd-even transposition sort. Normally, odd-even transposition sort has a while loop that is executed exactly *num_procs* times. In order to compare our sorting algorithms we make the while loop stop when the array is sorted.

ALGORITHM 8.1.

procedure *Odd_Even_Transposition_Sort(pi);*
singular *step, all_done;*
plural *pi, left, right, done;*
step $\leftarrow$ *0; all_done* $\leftarrow$ *False;*
while not *all_done* **do**
  step $\leftarrow$ *step + 1;*
  left $\leftarrow$ *broadcast_right(pi);*
  if *proc_id = 1* **then** *left* $\leftarrow -\infty$*;*
  right $\leftarrow$ *broadcast_left(pi);*
  if *proc_id = num_procs* **then** *right* $\leftarrow \infty$*;*
  done $\leftarrow$ *pi* $<$ *right;*
  if *proc_id* $\equiv$ *step mod 2* **then**
    if *left* $>$ *pi* **then** *pi* $\leftarrow$ *left;*
  else
    if *right* $<$ *pi* **then** *pi* $\leftarrow$ *right;*
  endif
  if not *done* **then** *done* $\leftarrow$ *broadcast_left(done);*
  all_done $\leftarrow$ *sing(1,done);*
endwhile

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. This is done by making each processor that detects an out of order value to its right broadcast that information to its left. As a result processor 1 learns if more sorting has to be done and informs the front-end. A total of three broadcast communication instructions are used to implement each step of odd-even insertion sort.

The following is a program for left adaptive insertion sort.

ALGORITHM 8.2.

procedure *Left_Adaptive_Insertion_Sort(pi);*
singular *all_done;*
plural *pi, left, right, data, pre_act, act, left_act;*
all_done $\leftarrow$ *False;*
while not *all_done* **do**
  left $\leftarrow$ *broadcast_right(pi);*
  if *proc_id = 1* **then** *left* $\leftarrow -\infty$*;*
  right $\leftarrow$ *broadcast_left(pi);*
  if *proc_id = num_procs* **then** *right* $\leftarrow \infty$*;*
  pre_act $\leftarrow$ *(left* $>$ *pi* **and** *right* $>$ *pi);* (A)
  data $\leftarrow \infty$*;*
  if *pre_act* **then** *data* $\leftarrow$ *broadcast_left(pi);*
  act $\leftarrow$ *pi* $<$ *data* **or** *pre_act;* (B)
  left_act $\leftarrow$ *True;*
  left_act $\leftarrow$ *broadcast_right(act);*
  if not *act* **and** *left_act* **then** *pi* $\leftarrow$ *data;*
  if not *(act* **or** *left_act)* **or** *pre_act* **then** *pi* $\leftarrow$ *left;*
  all_done $\leftarrow$ *sing(1,data)* $= \infty$*;*
endwhile

The variable names in the program match fairly closely the description of the left adaptive sorting strategy described in section 6. The set of pre-active processors are those which compute the value of *pre_act* equal to *True* on line (A). The blocking processors are those which have *pi* $<$ *data* in the calculation of *act* on line (B). The active processor are those that compute *act* equal to *True* on line (B). There is a slight exception to this in that the processors to the right of the right most pre-active processor also compute *act* equal

to *True*. Allowing these processors to be active does not change the result of a left adaptive insertion step. To complete the insertion step, a processor which is not active and whose left neighbors is active accepts the insertion stored in *data*, and 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*. Termination is achieved by setting the plural variable *data* to ∞. If processor 1's *data* remain ∞ after the pre-active processors broadcast, then the array must be sorted because there were no pre-active processors.

The two sorting algorithms are roughly the same length and complexity. The odd-even transposition sort uses two local communications and one long distance broadcast per iteration, while left adaptive insertion sort uses three local communications and one long distance broadcast per iteration. The main advantage of left insertion sort over odd-even transposition sort is that on some inputs the former has many fewer iterations than the latter. As we have shown in section 4, on average left adaptive insertion sort has only a slight advantage over odd-even transposition sort.

9 Two-way Sorting Strategies

We have some preliminary results on two-way sorting strategies, strategies which may employ both left and right insertion steps. Both the left greedy sort and left adaptive insertion sort can be modified into two-way sorting strategies. One modification is to simply alternate left insertion step and right insertion steps. We call these two methods *alternating greedy sort* and *alternating adaptive insertion sort*, respectively. Another modification is to simply choose the best, in terms of the maximum number of inversions removed, of a left or right insertion step. We call these two methods *greedy-greedy sort* and *greedy-adaptive insertion sort*, respectively. Interestingly, we have observed empirically that all these sorting strategies have about the same expected performance, namely, the average number of steps to sort is slightly larger than $n/2$. This is a two-fold speed-up over the one-way sorting algorithms. The maximum distance of values from their destinations in no longer a bottleneck for the two-way sorting strategies. Using a bandwidth argument we can show that the expected number of steps to sort with a two-way sorting strategy is at least $n/4$. An extension of the argument demonstrates that any alternating two-way sorting strategy has expected number of steps to sort at least $n/2$. It is most likely that any efficiently implementable two-way sorting strategy will be alternating. Hence, the alternating adaptive insertion sort, which is efficiently implementable, appears to be the best or nearly the best possible efficiently implementable sorting strategy for the one-dimensional sub-bus array.

10 Conclusion

We have shown that the maximum distance a value is from its destination in a data array is the main bottleneck for one-way sorting algorithms. We have given two optimal left-only sorting strategies, the left greedy and the left adaptive insertion. The left greedy sort is not efficiently implementable while the left adaptive insertion sort is.

There are a number of open problems left to solve. Is there a simple way to characterize optimal two-way sorting strategies such as the maxdist_L characterization of optimal left-only sorting strategies? Is the alternating adaptive insertion sort a near optimal two-way sorting strategy? Are four communication operations per left insertion step necessary to implement an optimal left-only sorting strategy or can it be done with three? Finally, how can the sub-bus capability be of advantage in two-dimensional sorting?

11 Acknowledgements

We wish to thank Martin Tompa for reading and commenting on early versions of this paper.

References

[1] T. Blank. The MasPar MP-1 architecture. 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. Technical Report 93-04-15, University of Washington, Department of Computer Science and Engineering, 1993. To appear in *SIAM Journal on Computing*.

[3] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, 1969.

[4] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 1973.

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

[6] 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*, pages 903–908, 1986.

[7] 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*, pages 255–263, 1986.