Randomized Sorting on the POPS Network*

Jaime Davila and Sanguthevar Rajasekaran

July 29, 2004

Abstract

Partitioned Optical Passive Stars (POPS) network has been presented recently as a desirable model of parallel computation. Several papers have been published that address fundamental problems on this architecture. The algorithms presented in this paper are valid for $POPS(d, g)$ where $d > g$ and use randomization. We present an algorithm that solves the problem of sparse enumeration sorting of $d^e$ keys in $\tilde{O}(\frac{d}{g}) + \log g)$ time and hence performs better than previous algorithms. We also present algorithms that allow us to do stable sorting of integers in the range $[1, \log n]$ and $[1, \frac{d}{g}]$ in $\tilde{O}(\lceil \frac{d}{g} \rceil + \log n)$ time. When $g = n^\epsilon$, for any constant $0 < \epsilon < \frac{1}{2}$ this allows us to do sorting of integers in the range $[1, n]$ in $\tilde{O}(\frac{1}{1-2\epsilon}(\lceil \frac{d}{g} \rceil + \log g))$ time. We finally use these algorithms to solve the problem of multiple binary search in the case where we have $d^e$ keys in $\tilde{O}(\frac{d}{g} + \log g)$ time and in the case where we have integer keys in the range $[1, n]$ in $\tilde{O}(\frac{1}{1-2\epsilon}(\lceil \frac{d}{g} \rceil + \log g))$ time, when $g = n^\epsilon$, for some constant $0 < \epsilon < \frac{1}{2}$.

*This research has been supported in part by the NSF Grants CCR-9912395 and ITR-0326155.
1 Introduction

A partitioned optical passive star network with parameters \( d \) and \( g \), \( POPS(d, g) \), is a parallel computational model that was introduced in [1],[5]. In a \( POPS(d, g) \) there are \( n = dg \) processors. These processors are divided onto \( g \) groups with \( d \) nodes in every group. There is an optical coupler between every pair of groups, so that at any given time step any coupler can receive a message from a processor of the source group and broadcast it to the processors of the destination group. The coupler which connects processors in group \( j \) (the origin group) with processors in group \( i \) (the destination group) will be labelled \( c(i, j) \). The \( j \)-th processor of the \( i \)-th group will be denoted as \( G_i(j) \) or equivalently \( P_{(i-1)d+j} \).

Each processor has \( g \) transmitters and is associated with \( g \) couplers, one for each of the \( g \) couplers that its groups is connected to, so that there are \( ng \) transmitters and receivers in such a network. This implies that the diameter of such networks is 1, since a message can be sent from any source processor to a destination processor using the coupler that connects the two corresponding groups. Another important feature of a \( POPS(d, g) \) is that because \( g \) messages can be sent from each group, the network bandwidth is \( g^2 \).

Several fundamental algorithms have been devised for \( POPS \) networks including for problems such as prefix calculation [3], [15], offline routing [9], [15], [14], online routing [3], [10] and matrix multiplication [14].

Sorting algorithms for this model can be obtained by the hypercube simulation described in [15] or by customized algorithms like the ones in [2]. The complexity of these algorithms is \( O(\lceil \frac{d}{g} \rceil \log^2 n) \) or \( O(\lceil \frac{d}{g} \rceil \log^2 g) \).
Figure 1: A $POPS(3, 2)$ network.
All the algorithms in this paper are designed for the $POPS(d, g)$ for the case $d > g$. In general the relationship between $d$ and $g$ is arbitrary, but the case $d > g$ seems more reasonable as has been pointed out in [2]. Additionally, in some of the algorithms we will assume that $g = n^\epsilon$, for any constant $0 < \epsilon < \frac{1}{2}$. This assumption will be stated explicitly whenever it is made.

We present algorithms for sparse enumeration sorting and integer sorting in a restricted range. Our randomized algorithms run in time $\tilde{O}(\lceil \frac{d}{g} \rceil + \log n)$. This run time is better than that of known algorithms.

The amount of resource (like time, space, etc.) used by a randomized algorithm is said to be $\tilde{O}(f(n))$ if the amount of resource used is no more than $c \alpha f(n)$ with probability $\geq (1 - n^{-\alpha})$, for all $n \geq n_0$. Here $c$ and $n_0$ are constants and $\alpha$ is any constant $\geq 1$. By high probability we mean a probability of $\geq (1 - n^{-\alpha})$, for any constant $\alpha \geq 1$.

In Section 2 we introduce a few preliminaries and algorithms that will be used later on. In Section 3 we present a randomized algorithm for sorting $d^\epsilon$ keys (where $0 < \epsilon < 1$ is a constant) that runs in time $\tilde{O}(\lceil \frac{d}{g} \rceil + \log g)$. In Section 4 we present algorithms for stable-sorting integers in a restricted range in $\tilde{O}(\lceil \frac{d}{g} \rceil + \log n)$ time. This allows us to develop algorithms for stable-sorting integers in the range $[1, n]$ in $\tilde{O}(\lceil \frac{d}{g} \rceil \frac{\log n}{\log \log n} + \frac{\log^2 n}{\log \log n})$ time and in $\tilde{O}(\frac{1}{\epsilon^2} (\lceil \frac{d}{g} \rceil + \log g))$ time when $g = n^\epsilon$ (for any constant $0 < \epsilon < \frac{1}{2}$). Finally in Section 5, we apply our sorting algorithms to solve the problems of Multiple Binary Search and IP routing.
2 Preliminaries

Definition 2.1 Let \( a_1, a_2, \ldots, a_n \) be a sequence of elements from some domain \( \Sigma \). Let \( \oplus \) be a binary, and associative operation defined on \( \Sigma \). Then, the prefix computation problem is defined to be that of computing the sequence \( a_1, a_1 \oplus a_2, \ldots, a_1 \oplus a_2 \oplus \ldots a_n \).

The following Lemma has been proven in [3] in the case of unit time computable operations:

Lemma 2.1 A POPS\((d, g)\) network can solve the prefix computation problem in \( O(\lceil \frac{4}{g} \rceil + \log g) \) slots (i.e., time).

The following Lemma also appears in [3]: (The proof of this Lemma is based on binary search).

Lemma 2.2 Let \( S_1 \) and \( S_2 \) be \( n \) key sequences, with one key of each per processor. If \( S_1 \) is sorted, a POPS\((d, g)\) network can find the rank of each key of \( S_2 \) in \( S_1 \) in a total of \( O(\lceil \frac{d}{g} \rceil + \log g) \) time.

Definition 2.2 Let \( a_1, a_2, \ldots, a_n \) be a sequence of \( n \) keys that are located one per processor in a POPS\((d, g)\) network. Each one of these keys has a destination processor associated with it. Let these destinations be \( d_1, d_2, \ldots, d_n \), respectively. The problem of routing is to send \( a_i \) to processor \( d_i \), for \( i = 1, 2, \ldots, n \). If the destinations are distinct, the problem of routing is known as permutation routing.

Since there are \( g^2 \) couplers in a POPS\((d, g)\) network, this implies that at least \( \frac{n}{g^2} = \frac{d}{g} \).
time is needed for routing a general permutation. The following result was proven in [10] and gives a nearly optimal algorithm for routing using randomization.

**Lemma 2.3** A $POPS(d, g)$ can solve the permutation routing problem in $\tilde{O}(\frac{d}{g} + \log \log g)$ time.

The previous algorithm is based on the two phase routing scheme proposed by Valiant and Brebner [17]. In this scheme, each packet is first sent to a random destination, by sending it to a random coupler and then to a random processor which listens to this coupler, and then from this intermediate destination the packet is sent to the actual destination.

The following simulation result of a hypercube on the POPS network is due to Sahni [15].

**Lemma 2.4** Each step of an $n$-node SIMD hypercube can be simulated in one slot on a $POPS(d, g)$ network (with $n = dg$), provided $d = 1$. If $d > 1$, then each step of the SIMD hypercube can be simulated in $2\lceil \frac{d}{g} \rceil$ slots on the $POPS(d, g)$ network.

**Definition 2.3** Let $a_1, a_2, \ldots, a_n$ be a sequence of keys that are located one per processor in a $POPS(d, g)$ network. The sorting problem is that of ordering these keys according to the group number and index of the processor inside the group. That is, $G_i(j) < G_k(l)$ iff $i < k$ or $(i = k$ and $j < l)$.

**Definition 2.4** The problem of sparse enumeration sorting is that of sorting $l$ keys in a network of size $n$ when $l < n$. Typical values of interest for $l$ will be $n^\epsilon$, for some constant $\epsilon > 0$. 

6
In this paper we are interested in sparse enumeration sorting and its applications on the POPS Network. Algorithms for general sorting (i.e., sorting \( n \) keys on an \( n \)-node POPS network) can be obtained by simulating hypercube algorithms. For instance, Batcher’s algorithm when implemented on the POPS network runs in time \( O(\lceil d/g \rceil \log^2 n) \). An algorithm with a run time of \( O(\lceil d/g \rceil \log^2 g) \) has been presented in [2].

The following Lemma on sparse enumeration sorting has been proven in [10] in the context of randomized selection:

**Lemma 2.5** A POPS(\( d, g \)) network can sort \( g \) keys in \( \tilde{O}(\lceil d/g \rceil \log g) \) time.

### 3 Sparse Enumeration Sorting

In this section we improve the bounds of Lemma 2.5 using randomization.

The basic idea to do sorting in this section is that of sampling as depicted by Frazer and McKellar [4]. This idea has been used extensively in the past to devise sequential and parallel randomized sorting algorithms. A brief description of the technique follows:

1. Randomly sample some keys from the input.
2. Sort the sample (using an appropriate algorithm).
3. Partition the input using the sorted sample as splitter keys.
4. Sort each part separately in parallel.

One key observation regarding the size of each of the parts considered in step 3 is given in the following Theorem [11]:

7
Lemma 3.1 Let $X = x_1, \ldots, x_n$ be a given sequence of $n$ keys and let $S = l_1, l_2, \ldots, l_s$ be a random sample of keys picked from $X$ in sorted order. Let $X_1 = \{l \in X : l \leq l_1\}$, $X_j = \{l \in X : l_{j-1} < l \leq l_j\}$ (for $2 \leq j \leq s$), and $X_{s+1} = \{l \in X : l > l_s\}$. The cardinality of each $X_j$ (for $1 \leq j \leq (s + 1)$) is $\tilde{O}(\frac{n}{s} \log n)$.

To perform step 4) of the algorithm, we partition the processors of the POPS network as follows.

**Definition 3.1** Given a $\text{POPS}(d, g)$ network, we divide every group (of size $d$) into subgroups of size $\frac{d}{g}$ each so that there will be $g$ such subgroups in a given group. We denote the $i$-th subgroup of the $j$-th group by $S^i_j$. The $k$-th element of $S^i_j$ will be denoted by $S^i_{j}(k)$.

**Lemma 3.2** Assume that $d$ elements are distributed among the processors of the subgroups $S^i_1, i = 1, \ldots, g$. A $\text{POPS}(d, g)$ can route these packets such that processor $S^i_j(k)$ gets the elements from the pair $(S^i_1(k), S^i_1(i))$ in $O(\lceil \frac{d}{g} \rceil)$ time.

**Proof:** We proceed in two phases. In the first phase we broadcast copies of subgroup $S^i_1$ among all the groups. In the second phase we route the values of $S^i_1(i)$ to the appropriate processor in each subgroup.

For the first phase we will proceed in $\frac{d}{g}$ stages. In the $k$-th stage we send the value of $S^i_1(k)$ to $S^i_1(k)$ and then we broadcast the value to $S^i_j(k)$, $j \neq i$. This could be done by the following routing step:

$$S^i_1(k) \rightarrow c(i, 1) \rightarrow S^i_1(k) \rightarrow c(j, i) \rightarrow S^i_j(k) \text{ for } j = 1, \ldots, g, j \neq i$$

In the second phase send the value of $S^i_j(j)$ to the group $S^i_j$ by using the following routing step:
\[ S_j^i(j) \rightarrow c(j,i) \rightarrow S_j^i(j) \rightarrow c(i,j) \rightarrow S_j^i \]

\[ \square \]

\[
\begin{array}{c|c|c}
(S_1^1(1), S_1^1(1)) & (S_1^2(1), S_1^2(2)) & (S_1^0(1), S_1^0(g)) \\
(S_1^1(\frac{d}{g}), S_1^1(1)) & (S_1^2(\frac{d}{g}), S_1^2(2)) & (S_1^0(\frac{d}{g}), S_1^0(g)) \\
(S_2^1(1), S_2^1(1)) & (S_2^2(1), S_2^2(2)) & (S_2^0(1), S_2^0(g)) \\
(S_2^1(\frac{d}{g}), S_2^1(1)) & (S_2^2(\frac{d}{g}), S_2^2(2)) & (S_2^0(\frac{d}{g}), S_2^0(g)) \\
(S_g^1(1), S_g^1(1)) & (S_g^2(1), S_g^2(2)) & (S_g^0(1), S_g^0(g)) \\
(S_g^1(\frac{d}{g}), S_g^1(1)) & (S_g^2(\frac{d}{g}), S_g^2(2)) & (S_g^0(\frac{d}{g}), S_g^0(g)) \\
\end{array}
\]

Figure 2: Illustration of lemma 3.2

**Theorem 3.1** Let \( l = d^\epsilon \), with \( 0 < \epsilon < 1 \). A POPS\((d,g)\) can sort \( l \) keys in \( \tilde{O}(\lceil \frac{d}{g} \rceil + \log g) \) time.

**Proof:**

The idea will be to follow the strategy presented at the beginning of this section by means of the following algorithm:

1. Each key decides to include itself in the sample with probability \( \frac{d}{2g} \).
2. Compact the sample keys in the first group, using a prefix computation.
3. Sort the sample keys.

9
4. Send each of the sample keys to every coupler, using the following routing step:

\[ G_1(k) \rightarrow c(k, 1) \rightarrow G_k(1) \rightarrow c(i, k) \text{ for all } k \neq i \]

5. Each of the keys in the first group does a binary search over the sample keys which are in every coupler, and determines to which \(X_i\) (according to lemma 3.1) it belongs.

6. Divide the first group into \(g\) subgroups of size \(\frac{d}{g}\) each. Let these subgroups be \(S_1^j\), \(j = 1, \ldots, g\). Keys that belong to \(X_i\) are routed to a random processor in \(S_1^j\).

7. Use Lemma 3.2 to perform a routing such that \(S_1^i(k)\) contains the pair of elements from \((S_1^i(k), S_1^i(i))\).

8. Every processor performs the comparison between the two values that it stores. Based on the comparison it generates a 0 (if the first key is greater than the second) or a 1 (otherwise).

9. We do a prefix computation on the bits generated in the previous step to calculate the rank of each key.

10. We route the keys based on their ranks.

We analyze the steps of the algorithm now.

In steps 1 and 2 we are selecting a sample of splitter keys and collecting these in the first group. The number of sample keys \(S\) has a binomial distribution \(B(d^k, \frac{d}{2g})\) whose mean is \(\frac{d^2}{2}\). By using Chernoff bounds we have that \(\Pr[|S - \frac{d^2}{2}| > \epsilon g] < e^{-\epsilon^2 g} \leq n^{-c\epsilon^2}\), for \(g \leq \log n\), so we can infer that the number of keys is \(\frac{d^2}{2} + o(g)\).

Since the number of keys of this sample is \(\frac{d^2}{2} + o(g)\) we know that we can apply lemma 2.5, and hence we can sort the set of the splitters.
In steps 4, 5 and 6 we split the keys according to the splitters. By using lemma 3.1 we know that the size of each set will be $\tilde{O}(g^\epsilon \log n)$, hence by picking an appropriate $\epsilon'$ we will have that the size of each set will be $g^{\epsilon'}$ with high probability. Notice also that since the number of keys is less than $d$ we are able to store these keys in the first group.

In steps 7, 8, 9 and 10 we sort the keys which are in the first group by calculating the rank of each one of them in the corresponding subgroup. Notice that since the size of each subgroup is $g^{\epsilon'}$, with $0 < \epsilon' < 1$, the arrangement produced by lemma 3.2 allows us to calculate the ranks of every key in its corresponding subgroup.

In order to account for the time complexity, we have that it follows from the following complexities of every step:

1. It takes $O(1)$ time.

2. It takes $O(\left\lceil \frac{d}{g} \right\rceil + \log g)$ time using lemma 2.1.

3. It takes $O(\left\lceil \frac{d}{g} \right\rceil + \log g)$ time employing lemma 2.5.

4. It takes $O(1)$ time.

5. It takes $O(\log g)$ time since we are doing a binary search over $g$ elements.

6. It takes $\tilde{O}(\left\lceil \frac{d}{g} \right\rceil + \log \log g)$ time in accordance with lemma 2.3.

7. It takes $O(\left\lceil \frac{d}{g} \right\rceil)$ time as per lemma 3.2.

8. It takes $O(1)$ time.

9. It takes $O(\left\lceil \frac{d}{g} \right\rceil + \log g)$ time using lemma 2.1.

10. It takes $\tilde{O}(\left\lceil \frac{d}{g} \right\rceil + \log \log g)$ time using lemma 2.3.
4 Integer Sorting

Suppose we are given \( n \) numbers where each number is an integer in the range \([1, n]\). The problem of sorting these numbers is known to be solvable in linear sequential time by using Radix Sorting \([7]\). Optimal parallel algorithms are also known for this problem on various parallel models (see e.g., \([11]\), \([13]\), \([12]\), and \([16]\)).

In order to develop the algorithms of this section, we introduce the following definition and lemmas, whose ideas are based on \([12]\).

**Definition 4.1** Let \( a_1, a_2, \ldots, a_n \) be input keys for a sorting algorithm, and let \( a_{\sigma(1)}, a_{\sigma(2)}, \ldots, a_{\sigma(n)} \) be the output of such an algorithm. We say that a sorting algorithm is stable if equal keys remain in the same relative order in the output as they were in the input. That is, if \( a_i = a_{i+j} \) then \( \sigma(i) < \sigma(i+j) \), for \( i = 1, \ldots, n, \ j > 0 \).

**Lemma 4.1** Let \( a_1, a_2, \ldots, a_n \) be a given input sequence, with \( a_i \in \{0, 1\} \) for \( i = 1, \ldots, n \).

A \( \text{POPS}(d, g) \) can stable sort these numbers in \( \tilde{O}(\left\lceil \frac{d}{g} \right\rceil + \log g) \) time.

**Proof:**

Consider the following algorithm:

1. Do a prefix-sum calculation over the keys \( a_i \). Let \( s_i \) be the value computed by the \( i \)-th processor.
2. Call $\tilde{a}_i$ the bit complement value of each key. Do a prefix-sum calculation over the values $\tilde{a}_i$. Let $\tilde{s}_i$ be the value computed by the $i$-th processor.

3. Broadcast to all processor the value $\tilde{s}_n$ (the number of zero-keys).

4. If the value of $a_i = 0$ we route that key to position $\tilde{s}_i$. If the value of $a_i = 1$ we route that key to position $\tilde{s}_n + s_i$.

It is clear that this algorithm is correct and that equal keys remain in the same relative order in the output as they were in the input, hence the algorithm is stable.

To prove the claimed complexity we just have to notice that steps 1 and 2 are prefix operations which can be done in $O(\lceil \frac{d}{g} \rceil + \log g)$ by using lemma 2.1.

Step 3 is a broadcast operation performed by the $n$-th processor and hence takes $O(1)$ time.

Finally, in step 4 we can use lemma 2.3, which gives us $\tilde{O}(\lceil \frac{d}{g} \rceil + \log g)$ time. □

For a proof of the following Lemma see [11]:

**Lemma 4.2** If the sequence $a_1, a_2, \ldots, a_n$ (where each $k_i \in [0, R]$ for $i = 1, \ldots, n$) can be stable sorted using $P$ processors in time $T$, then we can also stable sort $n$ numbers in the range $[0, R^c]$ in $O(T)$ time.

**Theorem 4.1** Let $a_1, \ldots, a_n$ be $k$-bit integers (for any fixed $k$). A $POPS(d, g)$ can sort these numbers in $\tilde{O}(k(\lceil \frac{d}{g} \rceil + \log g))$ time. Equivalently a $POPS(d, g)$ can do stable-sort of integers in the range $[1 \ldots n]$ in $\tilde{O}(\lceil \frac{d}{g} \rceil \log n + \log g \log n)$.

**Proof:** By using lemmas 4.1 and 4.2 the result follows immediately. □
By following the ideas used in [11] we will be able to state a result similar to lemma 4.1 for integers in the range \([1 \ldots h]\). To do that we need to introduce the following routing lemma.

**Lemma 4.3** Let \(a_1, a_2, \ldots, a_n\) be an input sequence such that processor \(P_i\) contains element \(a_i\) and let \(0 < h \leq d\). A \(POPS(d, g)\) can route these values so that processor \(P_{ih+q}\) has the values \((a_{ih+1}, \ldots, a_{(i+1)h})\), for \(i = 1, \ldots, \left\lfloor \frac{n}{h} \right\rfloor\) and \(q = 1, \ldots, h\) in \(O(\frac{d}{g} + h)\) time.

**Proof:** For simplicity of the proof we will assume \(h\) divides \(d\) and we will use the notation introduced in definition 3.1.

The algorithm will proceed in \(\frac{d}{g}\) stages. In stage \(m\) of the algorithm we will be sending the value of \(S_i^j(m)\) (\(1 \leq i \leq g\) and \(1 \leq j \leq g\)) to its destination processors \(G_i \left(\left\lfloor \frac{j + \frac{d-m}{h}}{h} \right\rfloor + q\right)\) for \(q = 1, \ldots, h\), in such a way that there are no collisions.

To do that we need to consider two cases. If \(g \leq \frac{d}{h}\) we have that no two messages originating from different subgroups \(S_i^j\) and \(S_i'^j\) will have the same destination. Hence we can route simultaneously \(g^2\) messages by the following two step routing:

\[
S_i^j(r) \rightarrow c(j, i) \rightarrow S_i'^j(i) \rightarrow c(i, j) \rightarrow G_i \left(\left\lfloor \frac{jd + mg}{gh} \right\rfloor + q\right),
\]

where \(i, j \in \{1, \ldots, g\}\) and \(q \in \{1, \ldots, h\}\).

So we obtain that if \(\frac{d}{g} \geq h\) we will expend \(O(\frac{d}{g})\) time.

When \(g > \frac{d}{h}\) we have that all the messages originating from \(S_i^j(r), S_i'^j(r)\) will have the same destination if \(\left\lfloor \frac{j}{d/h} \right\rfloor = \left\lfloor \frac{j'}{d/h} \right\rfloor\). In order to avoid these collisions, we will expend \(\frac{d}{d/h}\) steps. In step \(l\) we will send \(g^4\) messages simultaneously by using the following routing:

\[
S_i^j(r) \rightarrow c(j, i) \rightarrow S_i'^j(i) \rightarrow c(i, j) \rightarrow G_i \left(\left\lfloor \frac{jd + mg}{gh} \right\rfloor + q\right),
\]
where \( i, j \in \{1, \ldots, g\}, q \in \{1, \ldots, h\} \) and \( j \equiv l \mod \frac{d}{h} \).

So we have that if \( \frac{d}{q} < h \) the time we expend will be \( O\left(\frac{d}{q} \times \frac{2h}{q}\right) = O(h) \). This means that the total time expent is \( O(\max\{\frac{d}{q}, h\}) = O(\frac{d}{q} + h) \), hence the lemma follows.

\( \Box \)

**Lemma 4.4** Let \( a_1, a_2, \ldots, a_n \) be a given input sequence, with \( a_i \in [1, h] \) for \( i = 1, \ldots, n \) and \( h \leq d \). A POPS\((d, g)\) can stable sort these numbers in \( \tilde{O}\left(\lceil \frac{d}{g} \rceil + h + \log g\right) \) time.

**Proof:** We follow the approach used in **FineSort** of [11]. This strategy consists in giving each processor \( h \) successive keys. Each processor stable-sorts the keys given to it in a sequential way and then, collectively the processors group the keys with equal values. Finally the processors outputs a rearrangement of the keys in which first all the 1’s appear, then all the 2’s and finally all the \( h \)’s appear.

For simplicity of the proof and w.l.o.g we assume \( h \) divides \( n \). Consider the following algorithm where we have that \( i \in \{1, \ldots, \frac{n}{h}\} \) and \( j \in \{1, \ldots, h\} \).

1. Send the values \((a_{ih+1}, \ldots, a_{(i+1)h})\) to processors \( P_{ih+j} \).

2. Processors \( P_{ih+j} \) stable-sort the keys they have, by using a sequential algorithm, obtaining \((\tilde{a}_{ih+1}, \ldots, \tilde{a}_{(i+1)h})\). We define \( h \) lists per processor by
   \[
   L_{i,j} = \{a_l : a_l = j \text{ and } ih + 1 \leq l \leq (i+1)h \}.
   \]

   The elements in each list are ordered in the same relative order as in the input and the position of element \( \tilde{a}_{ih+j} \) in the list \( L(i, \tilde{a}_{ih+j}) \) is denoted by \( d(i, j) \).

3. Processors \( P_{ih+j} \) send the value \(|L(i, j)|\) to processors \( P_{(j-1)\frac{d}{h}+i} \).
4. The processors collectively perform the prefix sum of

\[ |L(0, 1)|, |L(1, 1)|, \ldots, |L(\frac{n}{h} - 1, 1)|, \]
\[ |L(0, 2)|, \ldots, \ldots, |L(\frac{n}{h} - 1, 2)|, \]
\[ \ldots, \ldots, \ldots, \ldots, \ldots, \]
\[ |L(0, h)|, \ldots, \ldots, |L(\frac{n}{h} - 1, h)| \]

We will call \( S(i, j) \) the prefix value computed by processor \( P_{(j-1)\frac{n}{h} + i} \).

5. Processors \( P_{(j-1)\frac{n}{h} + i} \) send the value \( S(i, j) \) to processor \( P_{ih+j} \).

6. Send the values \((S(i, 1), \ldots, S(i, h))\) to processors \( P_{ih+j} \).

7. Processors \( P_{ih+j} \) send the value \( b = \tilde{a}_{ih+j} \) to processor \( P_{S(i,b)+d(i,j)} \).

It is clear that the algorithm stable-sorts the given input. To see that it can do it within the claimed bounds we just have to notice that:

1. Steps 1 and 6 can be done in \( O(\frac{d}{g} + h) \) time by lemma 4.3.

2. Step 2 can be completed sequentially in time \( O(h) \).

3. Steps 3, 5 and 7 take \( \tilde{O}(\frac{d}{g} + \log \log g) \) time in accordance with lemma 2.3.

4. Step 4 takes time \( O(\frac{d}{g} + \log g) \) as per lemma 2.1.

\[ \square \]

The following are simple corollaries of lemma 4.4.

**Corollary 4.1** Let \( a_1, \ldots, a_n \) be integers in the range \([1, \log n]\). A \( POPS(d, g) \) can sort these numbers in \( \tilde{O}(\lceil \frac{d}{g} \rceil + \log n) \) time.
Corollary 4.2 Let \( a_1, \ldots, a_n \) be integers in the range \([1, \frac{d}{g}]\). A POPS\((d,g)\) can sort these numbers in \(\widetilde{O}(\lceil \frac{d}{g} \rceil + \log g)\) time.

Theorem 4.2 Let \( a_1, \ldots, a_n \), be integers in the range \([1 \ldots \log^c n]\). A POPS\((d,g)\) can stable sort these numbers in \(\widetilde{O}(c(\lceil \frac{d}{g} \rceil + \log n))\) time. This implies that a POPS\((d,g)\) can do stable-sort of integers in the range \([1 \ldots n]\) in \(\widetilde{O}(\lceil \frac{d}{g} \rceil \frac{\log n}{\log \log n} + \frac{\log^2 n}{\log \log n})\).

Proof: By using corollary 4.1 and lemma 4.2 the result follows. □

Theorem 4.3 Let \( a_1, \ldots, a_n \), be integers in the range \([1 \ldots n]\). A POPS\((d,g)\) where \(g = n^\epsilon\) and \(0 < \epsilon < \frac{1}{2}\) can stable sort these numbers in \(\widetilde{O}(\frac{\log n}{1-2\epsilon} (\lceil \frac{d}{g} \rceil + \log g))\) time.

Proof: By using corollary 4.2 and lemma 4.2 the result follows immediately. □

Theorem 4.4 Let \( a_1, \ldots, a_n \), be integers in the range \([1, n^{O(1)}]\). A POPS\((d,g)\) can stable sort these numbers in \(\widetilde{O} \left( \frac{\log n}{\log(d/g)} \left( \frac{d}{g} + \log g \right) \right)\) time.

Proof: By using corollary 4.2 the result follows immediately. □

5 Multiple Binary Search

Lemma 5.1 Let \( a_1, \ldots, a_n \) be a given sequence of keys stored one per processor in sorted order and let \( v \) be another input key. A POPS\((d,g)\) can perform a binary search of \( v \) on the input sequence in \(O(\lceil \frac{d}{g} \rceil)\) time.

Proof: The idea will be to use the following two step algorithm:
1. Send every value at any given processor to its neighbour processor.

2. Broadcast $v$. If $v$ falls between the two values that a processor holds, output the id of the processor.

The correctness of the algorithm is clear, and the time complexity is as claimed, because step 1 takes $O\left(\frac{d}{g}\right)$ time and step 2 takes $O(1)$ time. □

Now consider a generalization of binary search where we are given $d^\epsilon$ ($\epsilon$ being a constant $> 0$ and $< 1$) keys and one input sequence (of $n$ keys). The problem is to perform binary search in the input sequence for each of the $d^\epsilon$ keys.

**Theorem 5.1** A $POPS(d, g)$ can find the location of $d^\epsilon$, $0 < \epsilon < 1$ keys in a sorted sequence (one per processor) in $\tilde{O}(\left\lceil \frac{d}{g} \right\rceil + \log g)$ time.

**Proof:** Consider the following algorithm to do this operation:

1. Sort the $d^\epsilon$ keys using theorem 3.1. This will take us $O(\left\lceil \frac{d}{g} \right\rceil + \log g)$ time.

2. Calculate the ranks of the $d^\epsilon$ keys in the input sequence using lemma 2.2. This takes $O(\left\lceil \frac{d}{g} \right\rceil + \log g)$ time.

3. Route the keys according to their ranks using Lemma 2.3

□

Notice that we can use the strategy of the proof of theorem 5.1 by using in step 1 the algorithms given in theorems 4.1, 4.2, 4.3, and 4.4. This gives us the following results.

**Theorem 5.2** A $POPS(d, g)$ can find the location of $n$ integers in the range $[1\ldots \log^\epsilon n]$ in $\tilde{O}(e(\left\lceil \frac{d}{g} \right\rceil + \log g))$ time.
Theorem 5.3. A POPS($d, g$) where $g = n^\epsilon$, $0 < \epsilon < \frac{1}{2}$ can find the location of $n$ integers in the range $[1 \ldots n^\epsilon]$ in $\widetilde{O}(\frac{\epsilon}{1-2\epsilon}([\frac{d}{g}] + \log g))$ time.

Theorem 5.4. A POPS($d, g$) can find the location of $n$ integers in the range $[1, n^\epsilon]$ in $\widetilde{O}\left(\frac{\log n}{\log(d/g)} \left(\frac{d}{g} + \log g\right)\right)$ time.

Notice that multiple binary search has applications in IP-routing [8].

6 Conclusions

In this paper we have presented efficient randomized algorithms for sparse enumeration sorting of $d^\epsilon$, $0 < \epsilon < 1$ keys in $\widetilde{O}([\frac{d}{g}] + \log n)$ time and stable integer sorting in a fixed range in $\widetilde{O}([\frac{d}{g}] + \log g)$ time. We have also presented an algorithm for multiple binary search that runs in the same times bounds for $d^\epsilon$ keys and integer sorting. It is an interesting open question whether we can find an algorithm for general sorting that runs within the same time bounds.

7 Acknowledgments

The authors would like to thank the referees for their insightful comments.

References

[1] D. Chiarulli, R. Melhem, J. Teza, G. Gravenstreter, and S. Levitan, Multiprocessor interconnection networks using partitioned optical passive star (POPS) topologies and


