Sorting on Reconfigurable Meshes: 
An Irregular Decomposition Approach

TEN H. LAI* and MING-JYE SHENG†

Department of Computer and Information Science, The Ohio State University, Columbus, Ohio 43210

(Received 5 May 1997)

Most algorithms for reconfigurable meshes (R-meshes) are based on the divide-and-conquer (DAC) strategy. Although the strategy per se does not require the subproblems to be equal in size, existing DAC algorithms for R-meshes do divide the problem approximately evenly. This paper demonstrates that dividing a problem evenly is not necessarily a good way to decompose a problem. There are occasions on which an irregular decomposition scheme may be preferable. We take this approach and obtain a new sorting algorithm. Our sorting algorithm has several strengths: it is simple, scalable, and as broadcast-efficient as the best known result.

Keywords: Reconfigurable meshes, VLSI architectures, sorting

1. INTRODUCTION

A reconfigurable mesh (R-mesh) is a two- or three-dimensional array of processing elements (PEs) connected by reconfigurable buses. Though the PEs are interconnected as a regular mesh, the internal connection between the I/O ports of each PE can be individually reconfigured during the execution of algorithms [16]. Computations on reconfigurable meshes have recently received a great deal of attention from researchers. The challenge here is to solve a problem on an R-mesh in a constant number of steps using as few PEs as possible. Efficient algorithms have been developed for sorting [8, 9, 14, 18], image processing [5, 11, 12], arithmetic problems [6], computer vision [21] and geometric problems [7, 13, 19]. Almost all of these algorithms are based on the well-known divide-and-conquer (DAC) strategy: Decompose a given problem into several subproblems, solve them individually, and then combine the partial results into a solution to the original problem.

Although the divide-and-conquer method per se does not require subproblems to be equal or approximately equal in size, in practice almost all existing DAC algorithms follow this rule and divide a problem approximately evenly. A divide-and-conquer algorithm is said to follow a regular
decomposition scheme if it always divides a problem (approximately) evenly in size. Otherwise, it is based on an irregular decomposition scheme.

With regular decompositions, it is relatively easy to reduce problem size from \( n \) to \( o(n) \) in \( O(1) \) recursion steps, which is crucial in obtaining \( O(1) \) time algorithms. By comparison, there seems to be no easy control on subproblem size in irregular decompositions. Thus, intuitively, irregular decompositions seem unnatural and difficult to implement on R-meshes.

In this paper, we demonstrate that contrary to intuition, irregular decomposition can be a useful design paradigm.

We apply the irregular decomposition scheme to sort a set of \( n \) elements on an \( n \times n \) R-mesh. The resulting algorithm is conceptually simpler and computationally more efficient than previous sorting algorithms (e.g. [8, 9, 14, 18]). The irregular decomposition scheme can be tuned to use the same number of broadcasts as the best known result [22]. Our algorithm also has the desirable property of being scalable.

The purpose of this paper is thus twofold: (1) to demonstrate the usefulness of irregular decomposition, and (2) to report efficient sorting algorithms for R-meshes.

The next section reviews various models for reconfigurable meshes. The scheme of irregular decomposition is discussed in Section 3. It is then applied to obtain a new sorting algorithm. We also compare the new sorting algorithm with previous ones and present its scalability. All of the above can be found in Section 4.

2. ARCHITECTURES OF RECONFIGURABLE MESHES

2.1. The Model

A two-dimensional reconfigurable mesh (R-mesh for short) is a two-dimensional array of processors connected by a reconfigurable bus system. Each processor has four I/O ports, denoted as N (north), S (south), E (east), and W (west), respectively. An R-mesh with \( n \) rows and \( m \) columns of processors is denoted as an \( n \times m \) R-mesh. (A 6 \( \times \) 6 R-mesh is shown in the left portion of Fig. 1.) The rows of an R-mesh are numbered from 0 to \( n-1 \), with 0 referring to the northernmost row. Similarly, columns are numbered 0 to \( m-1 \), from west to east. \( PE(i,j) \) denotes the processor located at the intersection of row \( i \) and column \( j \). To simplify the notations, we use \( PE(i, *) \) (resp. \( PE(*, j) \)) to denote all processors in row \( i \) (resp. column \( j \)); and \( PE(*, *) \), all processors in the R-mesh. Each processor has a constant number of registers of \( O(\log n) \) bits. Data are represented in the binary form and stored in these registers. Buses are capable of carrying \( \Theta(\log n) \) bits data.

Internal connection between the four ports of each processor can be reconfigured during the execution of an algorithm. There are 15 different possible patterns of internal connection at each node (we use the notation \{xy\} to mean that port \( x \) and port \( y \) are connected to each other):

- no connections – {}
- Two-ports connections – \{NS\}, \{EW\}, \{NW\}, \{NE\}, \{SW\}, \{SE\}
- Three-ports connections – \{EW S\}, \{EW N\}, \{SN E\}, \{SN W\}
- Four-ports connections – \{EW SN\}
- Two-pairs connections – \{EW, SN\}, \{EN, WS\}, \{ES, WN\}.

These patterns of connection are pictorially shown in the right portion of Figure 1.

![Figure 1](image-url)
There are various models for reconfigurable meshes [1, 15, 16, 27, 28, 29]. They differ principally in the patterns of connection allowed at each processor:

- The **processor array with a reconfigurable bus system (PARBUS)** [27, 28] allows arbitrary connection between the ports.
- The **reconfigurable mesh with buses (RMESH)** [15, 16], or the **content addressable array processor (CAAP)** [29], allows all the connection patterns listed above except the two-pair connections.
- The **mesh restriction in reconfigurable network (MRN)** [1] allows all but three-ports and four-ports connections.

We will use **R-mesh** as a generic term to refer to any of the above models. When we describe an R-mesh algorithm, we will indicate which models of R-meshes we have in mind.

There are two models for bus arbitration in R-meshes [16]: **exclusive write** and **common write**. The **exclusive write model** does not allow bus conflict, *i.e.*, at most one processor can send data to a bus at any time. In this respect, it resembles the exclusive write PRAM (EW PRAM). The **common write model** allows multiple processors to simultaneously broadcast to the same bus as long as the same data value are broadcast. In this paper, we assume the exclusive write model.

In order to measure the time complexity of an R-mesh algorithm, two models have been proposed in [16]: the **unit-time delay model** and the **log-time delay model**. In the unit-time delay model, the computation operates synchronously, and the following operations are each counted as one step:

- evaluating an arithmetic or logic operation.
- reconfiguring internal connections between each port of a processor.
- broadcasting on a configured bus.

In the log-time delay model, broadcasting on a configured bus is assumed to take \( \log s \) time, where \( s \) is the diameter of the bus on which the data are broadcast [16]. Any algorithm taking \( T(n) \) steps on an \( n \times n \) R-mesh of the unit-time delay model will run in \( O(T(n) \log n) \) steps on the log-time delay model.

The unit-time delay model is more popular than the log-time delay model. In fact, almost all papers in the literature of R-mesh algorithms have assumed the unit-time delay model, which we will adopt in this paper, too.

### 3. THE IRREGULAR DECOMPOSITION SCHEME

Given a problem with an input set \( A \) (where \( |A| = n \)), a typical divide-and-conquer (DAC) R-mesh algorithm works as follows:

- **Divide** Partition \( A \) into \( A_0, A_1, \ldots, A_{m-1} \), and accordingly divide the given mesh \( M \) into submeshes \( M_0, M_1, \ldots, M_{m-1} \). (Assume this takes \( D(n) \) time.)
- **Conquer** Recursively solve each subproblem \( A_i \) on submesh \( M_i \) or directly solve \( A_i \) if the subproblem satisfies some boundary conditions. (The subproblems \( A_0, \ldots, A_{m-1} \) are solved simultaneously.)
- **Combine** Combine the partial results and obtain a solution to the original problem. (Assume this takes \( C(n) \) time.)

The time complexity \( T(n) \) of the algorithm satisfies the following recurrence relation:

\[
T(n) = D(n) + \max\{T(n_0), T(n_1), \ldots, T(n_{m-1})\} + C(n),
\]

where \( n_i = |A_i| \).

In order to have \( T(n) = O(1) \), a problem must be partitioned and subresults be combined both in \( O(1) \) time (i.e., \( D(n) = O(1) \) and \( C(n) = O(1) \)). Furthermore, each subproblem must be solved in \( O(1) \) time after \( O(1) \) recursion steps.

To satisfy these conditions, a problem must be carefully decomposed. A decomposition of a problem is said to be **regular** if the subproblems are approximately equal in size (i.e., \( |A_0| \approx |A_1| \approx \ldots \approx |A_{m-1}| \)). Otherwise, it is an **irregular** decomposition. A DAC algorithm is said to adopt
a regular decomposition scheme if it requires every decomposition to be regular. It adopts an irregular decomposition scheme if regular decomposition is not a requirement. Regardless of its regular or irregular status, a decomposition is said to be natural if a solution to the original problem can be easily obtained from the solutions to the subproblems (i.e., the “combine” step of the algorithm is trivial). For instance, the well-known (sequential) quicksort adopts an irregular decomposition scheme, while the mergesort does a regular one. The quicksort always produces natural decompositions (since the “combine” step takes only \(O(1)\) sequential time); the mergesort doesn’t (it takes \(O(n)\) time to merge two sorted subarrays).

Regular Decomposition

With a regular decomposition scheme on an R-mesh, a problem of size \(n\) is typically divided into a number \(m\) of subproblems each of size approximately \(n/m\). Correspondingly, the R-mesh is evenly divided into \(m\) submeshes such that each submesh accommodates a subproblem. As opposed to the binary partition scheme which is commonly used in existing parallel algorithms and requires \(O(\log n)\) recursive steps, only \(O(1)\) recursive steps are allowed in order to have \(D(n) = O(1)\). A common technique in existing R-mesh algorithms is to assign \(m\) as \(n^{\varepsilon_1}\) for some \(\varepsilon_1 < 1\), so that after at most \(O(1/\varepsilon_1)\) recursive steps, sizes of subproblems can be reduced to \(O(n^{\varepsilon_2})\) where subproblems of size \(O(n^{\varepsilon_2})\) are easy to compute in \(O(1)\) time. If the decomposition is not a natural one, then combining \(m\) partial results in \(O(1)\) time is not an easy task. Most existing algorithms for R-meshes uses \(m^2\) equal-sized submeshes in the “combine” step to achieve \(C(n) = O(1)\).

Irregular Decomposition

In an irregular decomposition scheme, the subproblems \(A_j\)'s are not necessarily of the same size, nor are the submeshes \(M_j\)'s. When both regular and irregular decomposition schemes are capable of producing natural decompositions, but the irregular scheme is computationally easier because of less restriction on subproblem sizes, irregular decompositions may be preferable. In this case, a problem of size \(n\) is divided into \(m\) subproblems of sizes \(n_0, \ldots, n_{m-1}\), with max \(\{n_0, n_1, \ldots, n_{m-1}\} \leq n^\varepsilon\) for some \(\varepsilon, 0 < \varepsilon < 1\). If \(C(n) = D(n) = O(1)\) and \(T(n^\varepsilon) = O(1)\) for some \(\varepsilon', 0 < \varepsilon' < 1\), then \(T(n) = 1\).

We shall demonstrate this case with sorting algorithms in the next section.

4. SORTING BY IRREGULAR DECOMPOSITION

Several optimal sorting algorithms for R-meshes are available in the literature, all based on the divide-and-conquer strategy with regular decompositions [9, 14, 18]. In this section, we apply the irregular decomposition scheme and obtain a new sorting algorithm that is simple and requires fewer bus broadcasts than previous algorithms.

Our algorithm is an improvement on the one by Lin et al. [14], so we start with a brief review of the latter algorithm.

4.1. A Regular Decomposition Approach

Let \(S\) be a set of \(n\) integers to be sorted. For ease of presentation, we assume the \(n\) elements to be distinct so as to avoid tedious bookkeeping details. Also, we assume that a number \(\infty\) is added to \(S\) so that \(S\) now contains \(n + 1\) numbers. The sorting algorithm presented in [14] divides the given set \(S\) into \(m\) subsets \(S_k\), \(0 \leq k \leq m - 1\), of the same size, such that all elements in \(S_i\) are less than all elements in \(S_{i+1}\). This is a “natural” decomposition because once each \(S_j\) is sorted, the whole set \(S\) is sorted. To sort the \(S_i\)'s, the given \(n \times n\) mesh is divided into \(m\) submeshes, and each \(S_i\) is sorted on a dedicated submesh. These ideas are more precisely stated in the following.

Lin–Olariu–Schwing–Zhang Algorithm

1. Select \(m + 1\) elements \(s_0, s_1, \ldots, s_m\), such that \(s_k\) is the \(q_k\)th largest element in \(S\), where \(m = \sqrt[3]{n}\) and \(q_k = k \times n/m + 1\).
2. Partition $S$ into $m$ subsets, $S_k$, $0 \leq k \leq m - 1$, such that $S_k = \{ s \in S | s_k \leq s < s_{k+1} \}$.
3. Directly sort each $S_k$ on an $n \times (n/m)$ submesh $M_k$.
4. Concatenate the sorted subsets into a single sorted list.

The algorithm employs a sophisticated multi-selection algorithm to carry out Step 1. The multi-selection algorithm, while clever and interesting in its own right, is undesirably complex and involved for the sorting problem, especially on an R-mesh.

4.2. An Irregular Decomposition Scheme

The above sorting algorithm requires equal size of the subproblems, and, therefore, needs to solve a relatively difficult multi-selection problem. We observe that dividing the problem evenly among the subproblems is not crucial for the above algorithm to work correctly. If we relax this requirement and allow the subproblems to be different in size, then we don’t need to solve the hard multi-selection problem and thus have a simpler and more efficient algorithm. Of course we still need some restriction on subproblem sizes so as to not partition a problem into a very large subproblem and some very small ones.

Our algorithm is a recursive one, so we describe it as one that sorts $n'$ elements on an $n \times n'$ R-mesh. (Initially we are given $n$ elements to sort on an $n \times n$ R-mesh.) The algorithm divides a problem of size $n'$ into $m$ subproblems of sizes $n_0, \ldots, n_{m-1}$, respectively, such that $\max\{n_0, n_1, \ldots, n_{m-1}\} \leq f(n')$, where

$$f(n') = \begin{cases} \frac{\sqrt{n'}}{2} & \text{if } n' \leq \frac{\sqrt{n'}}{2} \\ \frac{\sqrt{n'}}{2} & \text{otherwise} \end{cases}$$

Let $T(n')$ be the time complexity of our algorithm in sorting $n'$ elements on an $n \times n'$ R-mesh. The algorithm is specified in the following.

**New Sorting Algorithm: Sort($S$)**

**Input:** a set $S$ of $n'$ elements stored, one element per processor, in the northernmost row of an $n \times n'$ R-mesh.

**Output:** the set $S$, sorted in ascending order, again stored one element per processor in the northernmost row of the same R-mesh.

Define some variables for ease of reference:

$q = [\sqrt{n}]$, $p = [n'/\lceil f(n') \rceil]$, and $m = p \times q$.

1. If $n' \leq \sqrt{n}$ then directly sort the elements by the method of [28]. (It takes $O(1)$ time.)
2. Select a set $S'$ of $m + 1$ elements $s_0, s_1, \ldots, s_m$ from $S$ such that the number of elements $x$ in $S$ that satisfies $s_k \leq x \leq s_{k+1}$ is at most $f(n')$. Store the elements of $S'$ in the northernmost row of $M$. See procedure Select($S$) for implementation of this step. (Assume that this step requires $D(n')$ time.)
3. Partition $S$ into $m$ subsets $S_k$, $0 \leq k \leq m - 1$, such that $S_k = \{ s \in S | s_k \leq s < s_{k+1} \}$. Then route $S_k$ to a dedicated $n \times |S_k|$ submesh $M_k$. See procedure Partition($S, S'$) for details. (Assume that this step requires $R(n')$ time.)
4. Simultaneously sort $S_k$ on $M_k$ for all $k, 0 \leq k \leq m - 1$, recursively using this algorithm. (This step needs $T(f(n'))$ time.)
5. Combine the sorted subsets into a single sorted list. (It takes $C(n') = O(1)$ time.)

To carry out Step 2 of the algorithm, we perform the following procedure.

**Select($S$)**

1. Divide the mesh evenly into $p$ submeshes, $M_i, 0 \leq i \leq p - 1$, where the size of each $M_i$ is $n \times \lceil f(n') \rceil$ except for the last one, $M_{p-1}$, which may be narrower than others. This correspondingly divides $S$ into $p$ sublists, $W_i, 0 \leq i \leq p - 1$, with $W_i$ consisting of the elements of $S$ that are stored in $M_i$. Note that the elements in $W_i$ are not necessarily smaller than those in $W_{i+1}$. (This step obviously takes $O(1)$ time.)
2. Sort $W_i$ on $M_i$ for $0 \leq i \leq p - 1$, simultaneously. (This recursive step takes $T(f(n'))$ time.)
3. For each sorted $W_i$, select $w_{l,0}, w_{l,1}, \ldots, w_{l,q-1}$ such that these elements evenly partition $W_i$ into subsets of size $f(n')/q$. These selected elements are marked and stored in the northernmost row of $M$. (This step takes $O(1)$ time.)
4. Let $S' = \{ s_k : 0 \leq k \leq m-1 \}$ be the set of all the elements selected in the preceding step. Include $s_m = \infty$ in $S'$ for convenience in presentation.

The $s_k$'s ($0 \leq k \leq m-1$) so produced are to be used in Step 3 of the New Sorting Algorithm to divide the input set $S$ into subsets $S_k$, $0 \leq k \leq m-1$. The following lemma shows that these elements do have the property as required by the sorting algorithm.

**Lemma 1** If $S_k = \{ s \in S | s_k \leq s < s_{k+1} \}$, then $|S_k| \leq f(n')$.

**Proof** Note that $W_l$ can be divided by $w_{l,0}, w_{l,1}, \ldots, w_{l,q-1}$ into $W_{l,0}W_{l,1}, \ldots, W_{l,q-1}$. Also let

$$U_k = \{ s \in S | s \geq s_{k+1} \}.$$  

$$L_k = \{ s \in S | s < s_k \}.$$  

Consider the sequence $w_{l,0} \leq \ldots \leq w_{l,q-1}$ and its relation to $s_k$ and $s_{k+1}$. There are three possible cases:

- $w_{l,0} < w_{l,1} < \ldots < w_{l,q-1} \leq s_k < s_{k+1}$.
- $s_k < s_{k+1} < w_{l,0} < w_{l,1} < \ldots < w_{l,q-1}$.
- There exists an $l'$ such that $w_{l',0} < \ldots < w_{l',r} \leq s_k < s_{k+1} \leq w_{l',r+1} \ldots < w_{l,q-1}$.

In all cases, at most one set from $W_{l,0}, W_{l,1}, \ldots, W_{l,q-1}$ is not a subset of $U_k \cup L_k$. Since there are $p$ such sequences (any two of them are disjoint), we have

$$|U_k| + |L_k| \geq \sum_{w_{l',r+1} \leq W_{l',r} \leq s_{k+1}} (|W_{l,r}|) \geq n' - \lfloor f(n')/q \rfloor \ast p.$$  

Thus, $|S_k| = n' - (|U_k| + |L_k|) \leq \lfloor f(n')/q \rfloor \ast p$.

Since

$$\lfloor f(n')/q \rfloor \ast p \leq f(n')/q \ast p = f(n') \ast (p/q) \quad (1)$$  

and

$$p = \lceil n'/f(n') \rceil \leq \lceil n^{1/4} \rceil = q, \quad (2)$$  

it follows that $|S_k| \leq f(n')$.

Once $S'$ is computed, $S$ is partitioned by the elements in $S'$ into $m$ subsets $S_k$, $0 \leq k \leq m-1$ such that $S_k = \{ s \in S | s_k \leq s < s_{k+1} \}$; each $S_k$ is then moved to a dedicated $n \times |S_k|$ submesh. This can be done as follows.

**Partition $(S, S')$**

1. For each element in $S'$, compute its rank in $S$. (This can be done in parallel on an $n' \times n'$ R-mesh in $O(\log n'/\log (n' - \log m))$ time [14].)

2. If $s_k$ has rank $\pi(k)$, then route $s_k$ to $PE(0, \pi(k))$.

Divide the $n \times n'$ mesh into submeshes $M_0, M_1, \ldots, M_{m-1}$, with $M_k$ of size $n \times (\pi(k + 1) - \pi(k))$. (This obviously takes $O(1)$ time.)

3. Broadcast $S$ to the westernmost column of each $M_k$. (This takes $O(1)$ time.)

4. For each submesh $M_k$, mark the processors in its westernmost column for which the elements received from the previous broadcast are in $S_k$. Then extract those elements in the marked processors from its westernmost column to its northernmost row. (This can be done in $O(1)$ time by Lemma 2 below.)

**Lemma 2** Suppose each $PE(i,j)$ holds a value $a_j$ in an $r \times c$ R-mesh $M$, and out of the $c$ values $-a_0, \ldots, -a_{c-1}$ or fewer are marked. (If $a_j$ is marked, then all of its copies as stored in column $j$ are marked.) It can be done in one broadcast to extract the marked values and store them in the westernmost column.

**Proof** Configure the internal connection $A(i,j)$ of $PE(i,j)$ as follows:

$$A(i,j) = \begin{cases} \{WN, ES\} & \text{if } a_j \text{ is marked} \\ \{WE\} & \text{otherwise} \end{cases}$$

For example, if $a_1, a_4, a_5, a_7$ are marked, the R-mesh configuration is as illustrated in Figure 2.

After the reconfiguration, let each $PE(0,j)$ that holds a marked $a_j$ broadcast $a_j$ to its $N$-port. The value will be received by a $PE$ in the westernmost column.}

In parallel to Lemma 2, the following is true.
Corollary 1 Suppose each PE \((i,j)\) holds a value \(a_i\) in an \(r \times c\) R-mesh \(M\), and out of the \(r\) values \(- a_0, \ldots, a_{r-1}\) \(- c\) or fewer are marked. (If \(a_i\) is marked, then all of its copies as stored in row \(i\) are marked.) It can be done in one broadcast to extract the marked values and store them in the northernmost row.

Now we analyze the complexity of our sorting algorithm. The time complexity \(T(n')\) satisfies the following recurrence relations:

\[
f(n') = \begin{cases} \sqrt{n^2} & \text{if } \sqrt{n} < n' \leq \sqrt{n^3} \\ \sqrt{n^3} & \text{if } \sqrt{n^3} < n' \\ m = \lceil n'/\lceil f(n') \rceil \rceil \ast \sqrt{n} \\ D(n') + R(n') + T(f(n')) + C(n') \\ O(1) & \text{if } n' > \sqrt{n} \\ \end{cases}
\]

\[
T(n') = \begin{cases} T(\sqrt{n^3}) & \text{if } \sqrt{n^3} < n' \\ T(\sqrt{n}) & \text{if } \sqrt{n} < n' \leq \sqrt{n^3} \\ \end{cases}
\]

Thus, \(T(n) = O(1)\).

We summarize the above results as a theorem.

Theorem 1 The New Sorting Algorithm sorts a set of \(n\) elements on an \(n \times n\) R-mesh in \(O(1)\) time.

The R-mesh algorithm described above works on the PARBUS model. It can also be applied to the more restrictive models, RMESH and MRN. In the following, we discuss issues about implementing this sorting algorithm in RMESH and MRN.

First note that our algorithm employ an existing algorithm [28] to sort \(\sqrt{n}\) elements on an \(n \times \sqrt{n}\) R-mesh. The algorithm in [28] does not use any crossing connection \{WE, NS\}, 3-port connection, or 4-port connection. It does use non-crossing two-pair connections \{WN, ES\} and \{WS, EN\}. These connections can be simulated using four PEs in the RMESH model [6]. Thus, a set of \(\sqrt{n}\) elements can be sorted in \(O(1)\) time on \(n \times \sqrt{n}\) R-meshes (PARBUS, RMESH and MRN). Similarly, the same simulation can also be applied to compute Step 4 of the partition procedure.

Another implementation issue is the ranking algorithm of [14] (in Step 1 of the partition procedure). That algorithm needs to use the crossing connection \{WE, NS\}. Fortunately, computing the sum of a 0/1 sequence of length \(n\) on an \(n \times 2n^e\) R-mesh (PARBUS, RMESH, or MRN) can be done in \(O(1)\) time for \(0 < e < 5\). Using this algorithm, we can compute the sum of a 0/1 sequence of length \(n\) on an \(n \times n\) R-mesh (PARBUS, RMESH, and MRN) in \(O(n^e/(\log n^e - \log m))\) time.

4.3. Comparison with Existing Sorting Algorithms

The model employed in our analysis of time complexity takes into account both computation time and communication time. Each primitive computation as well as each reconfiguration of the bus system and the subsequent broadcast of data, is assumed to be completed in a unit time. However, the constant involved in a reconfiguration and broadcast is normally much larger than that involved in a computation step. In light of
this, Nigam and Sahni [18] proposed the number of bus broadcasts as a measure of efficiency of $O(1)$ time R-mesh algorithms. They showed that the number of broadcasts required by their sorting algorithms is much smaller than that of [14] and that of [9]. Table I summarizes the number of broadcasts needed by various sorting algorithms. As one can see from the table, our algorithm is more efficient than previous ones. Recently, Olariu and Schwing presented a deterministic sampling scheme that further reduces the number of broadcasts required for the sorting problem [22]. Our algorithm on the PARBUS can achieve the same number of broadcasts by setting the following parameters:

$$f(n') = \begin{cases} \sqrt[3]{n^3} & \text{if } n' \leq \sqrt[3]{n^3} \\ \sqrt[3]{n^3} & \text{otherwise} \end{cases}$$

and $q = \lceil \sqrt[3]{n^3} \rceil$. The reader is referred to Appendix A and Appendix B for analyses of the numbers of broadcasts required by our algorithm.

4.4. Scalability of the New Sorting Algorithm

A scalable sorting algorithm can sort a set of $n$ elements on an $n/K \times n/K$ R-mesh in $O(K)$ steps for $1 \leq K \leq \sqrt{n}$. This also leads to an $AT^2$ optimal design in the word model of VLSI. The Jang–Prasanna [9] algorithm is scalable. The Nigam–Sahni [18], Lin–Olariu–Schwing–Zhang [14], and Olariu–Schwing [22] algorithms seem to be not scalable, or at least hard to scale. Our algorithm, while close to the Lin et al., algorithm, is easy to scale.

| R-mesh Algorithms | Number of Broadcasts | PARBUS | RMESH |
|-------------------|----------------------|--------|-------|
| Olariu–Schwing [22] | 35 | not applicable | 75 |
| Our Original Algorithm | 35 | 75 |
| with new parameters | | |
| Nigam–Sahni [18] | 59 (Column Sort) | 120 (Rotate Sort) | |
| Lin–Olariu–Schwing–Zhang [14] | | not applicable | |
| Jang–Prasanna [9] | > 103 | not applicable | |

Although the Jang–Prasanna [9] algorithm is scalable, it only works on a PARBUS. Ours works for all R-mesh models (PARBUS, RMESH, and MRN). Our algorithm is also easier to implement; we only need an algorithm for $1 \leq K \leq \sqrt{n}$, while [9] needs two algorithms, one for $1 \leq K \leq \sqrt{n}$, and one for $\sqrt{n} \leq K \leq \sqrt{n}$.

Our algorithm divides the problem of size $n'$ into $m$ subproblems of sizes $n_0, n_1, \ldots, n_{m-1}$, respectively, such that $\max\{n_0, n_1, \ldots, n_{m-1}\} \leq f(n')$, where

$$f(n') = \begin{cases} n/\sqrt{K} & \text{if } n/\sqrt{K} < n' \\ n/K & \text{otherwise} \end{cases}$$

To sort the original set of $n$ elements, it divides $n$ elements recursively until the size of every subproblem is less than $n/K$. The subproblems are then solved in $O(1)$ time by Theorem 1.

We also use amortization [4] to prove the time complexity of this scalable algorithm. To the best of our knowledge, this is the first usage of amortization in the design and analysis of R-mesh Algorithms. The details of this algorithm and its time complexity analysis are described in Appendix C.

Theorem 2 The New Sorting Algorithm sorts a set of $n$ elements on an $n/K \times n/K$ R-Mesh in $O(K)$ time for $1 \leq K \leq \sqrt{n}$.

5. CONCLUDING REMARKS

We have presented an irregular decomposition technique in the design and analysis of R-mesh algorithms for sorting problems and come up with $O(1)$ time algorithms on reconfigurable meshes using $n^2$ processors. All algorithms in the paper, designed for a PARBUS, can be also run on RMESH and MRN.

Our sorting algorithms on a two dimensional R-mesh of size $n/K \times n/K$, where $1 \leq K \leq \sqrt{n}$, requires an $AT^2$ (area-time-square-product) upper bound of $O(n^2)$. Since an R-mesh can be realized using $n^2$ area, this is the best one can obtain in terms of the lower bound result $AT^2 = \Omega(n^2)$ for the sorting problems on the VLSI word model [26].
The R-mesh algorithm (using irregular decomposition) presented here, unlike previous R-mesh algorithms (using regular decomposition), looks a little complicated at first sight. In contrast to our intuition, it is very simple to implement on R-meshes, and as broadcast-efficient as the best known results obtained by [22].

We have applied irregular decomposition to obtain simple and efficient sorting algorithms for R-meshes. The main idea behind the notion of irregular decomposition is to divide a problem along its “natural” boundaries between subproblems. Not all problems have such natural boundaries. This paper provides an evidence that when a problem has natural boundaries between its subproblems, irregular decomposition may be a more effective way for designing a divide-and-conquer algorithm for R-meshes.

References

[1] Ben-Asher, Y., Peleg, D., Ramaswami, R. and Schuster, A. (1991). The power of reconfiguration, Journal of Parallel and Distributed Computing, 13, 139–153.
[2] Bokka, V., Gurla, H., Olariu, S. and Schwing, J. L. (1994). Constant time convexity problems on dense reconfigurable meshes, In: International Conference on Parallel Processing, pp. III-210–213.
[3] Chin, F. Y., Lam, J. and Chen, I. (1982). Efficient parallel algorithms for some graph problems, Communications of the ACM, 25(9), 659–665.
[4] Cormen, T. H., Leiserson, C. E. and Rivest, R. L. (1990). Introduction to Algorithms, McGraw-Hill Book Company.
[5] Jang, J., Park, H. and Prasanna, V. (1992). A fast algorithm for computing histogram on reconfigurable meshes, In: Symposium on the Frontier of Massively Parallel Computation.
[6] Jang, J., Park, H. and Prasanna, V. An optimal multiplication algorithm on reconfigurable meshes, In: IEEE Symposium on Parallel and Distributed Processing, December 1992.
[7] Jang, J. and Prasanna, V. (1992). Efficient parallel algorithms for some geometric problems on reconfigurable meshes, In: International Conference on Parallel Processing, pp. 127–130. CRC Press, Boca Raton, FL.
[8] Jang, J. and Prasanna, V. (1992). A fast sorting algorithm on higher dimension reconfigurable meshes, In: Proceedings 26th Conference on Information Sciences and Systems.
[9] Jang, J. and Prasanna, V. (1992). An optimal sorting algorithm on reconfigurable meshes, In: International Parallel Processing Symposium, pp. 130–137, IEEE Computer Society, Los Alamitos, CA.
[10] Jaromczyk, J. W. and Kowaluk, M. (1991). Constructing the relative neighborhood graph in 3-dimensional euclidean space, Discrete Applied Mathematics, 31, 181–191.
[11] Jenq, J. and Sahni, S. (1991). Reconfigurable mesh algorithms for image shrinking, expanding, clustering, and template matching, In: International Parallel Processing Symposium, pp. 208–215.
[12] Jenq, J. and Sahni, S. (1992). Histogramming on a reconfigurable mesh computer, In: International Parallel Processing Symposium, pp. 425–432.
[13] Lai, T. H. and Sheng, M. J., “Triangulations on Reconfigurable Meshes: A Natural Decomposition Approach,” Journal of Parallel and Distributed Computing, 30 (October 1995), 38–51.
[14] Lin, R., Olariu, S., Schwing, J. and Zhang, J. (1992). A VLSI-optimal constant time sorting on a reconfigurable mesh, In: Proceedings of Ninth European Workshop on Parallel Computing, pp. 1–16, Madrid, Spain.
[15] Miller, R., Prasanna Kumar, V. K., Reisis, D. and Stout, Q. F. (1988). Meshes with reconfigurable buses, In: Proceedings of the MIT Conf. on Advanced Research in VLSI, pp. 163–178.
[16] Miller, R., Prasanna Kumar, V. K., Reisis, D. and Stout, Q. F. (1993). Parallel computations on reconfigurable meshes, IEEE Transactions on Computers, pp. 678–692.
[17] Miller, R. and Stout, Q. F. (1989). Mesh computer algorithms for computational geometry, IEEE Transactions on Computers, C-38(3), 321–340.
[18] Nigam, M. and Sahni, S. (1993). Sorting n numbers on n x n reconfigurable meshes with buses, In: International Parallel Processing Symposium, pp. 174–181.
[19] Nigam, M. and Sahni, S. (1994). Computational geometry on a reconfigurable mesh, In: International Parallel Processing Symposium, pp. 258–261.
[20] Nigam, M. and Sahni, S. (1994). Triangulation on a reconfigurable mesh with buses, In: International Conference on Parallel Processing, pp. III-251–257.
[21] Olariu, S., Schwing, J. L. and Zhang, J. (1992). Fast computer vision algorithms for reconfigurable meshes, In: International Parallel Processing Symposium, pp. 258–261.
[22] Olariu, S. and Schwing, J. L. (1996). A Novel Deterministic Sampling Scheme with Application to Broadcast-Efficient Sorting on the Reconfigurable Mesh, In: Journal of Parallel and Distributed Computing, pp. 215–222.
[23] Preparata, F. P. and Shamos, M. I. (1985). Computational Geometry: An Introduction, Springer-Verlag, New York.
[24] Savage, C. and JaJa, J. (1981). Fast, efficient parallel algorithms for some graph problems, SIAM Journal on Computing, 10(4), 682–691.
[25] Trahan, J. L., Vaidyanathan, R. and Subbaraman, C. (1994). Constant time graph and poset algorithms on the reconfigurable multiple bus machine, In: International Conference on Parallel Processing, pp. III-210–213.
[26] Ullman, J. D. (1984). Computational Aspects of VLSI Computation, Computer Science Press.
[27] Wang, B. and Chen, G. (1990). Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems, IEEE Transaction on Parallel and Distributed Systems, 1(4), 500–507.
[28] Wang, B., Chen, G. and Lin, F. (1990). Constant time sorting on a processor array with a reconfigurable bus system, Information Processing Letters, 34(4), 187–190.
[29] Weems, C. C., Levitan, S. P., Hanson, A. R., Riseman, E. M., Nash, J. G. and Sheu, D. B. (1989). The image understanding architecture, International Journal of Computer Vision, 2, 251–282.
APPENDIX A: NUMBER OF BROADCASTS IN THE PARBUS MODEL

We describe our PARBUS implementation of the new algorithm in two lemmas, Lemma 3 for the non-recursive part and Lemma 4 for the recursive part.

Lemma 3 Given a set $S$ of $n'$ elements allocated in the processors of the northernmost row of an $n \times n'$ R-mesh $M$. If $n' \leq \sqrt{n}$, $S$ can be sorted and stored in the processors of the northernmost row of $M$ in 4 broadcasts.

Proof

Nonrecursive_Sort($S$)

1. First broadcast the elements in $S$ by column buses such that PE(*,j) has $a_j$. Then Partition $M$ into $\sqrt{n}$ submeshes, $M_0, M_1, \ldots, M_{\sqrt{n}-1}$, each of size $\sqrt{n} \times n'$.
2. For each submesh $M_k$, broadcast $a_k$ to each processor of $M_k$.
3. For each submesh $M_k$, each PE($i$, j) of $M_k$ has $a_k$ and $a_j$. We configure the internal connection $A(i,j)$ of PE($i$, j) as follows.

   $$A(i,j) = \begin{cases} \{WN, ES\} & \text{if } a_j \leq a_k \\ \{NS\} & \text{otherwise} \end{cases}$$

After the configuration, send $a_k$ from the N port of $PE(k \times \sqrt{n}, n' - 1)$ via the configured subbus. If $a_k$ has rank $\Pi(k)$, then $PE(k \times \sqrt{n} + \Pi(k), 0)$ is connected to the subbus, and marked.

For example, if $P(a_0) = P(a_2) = P(a_3) = \ldots P(a_6) = 0$ and $P(a_4) = P(a_5) = P(a_7) = 1$, then an R-mesh configuration is illustrated in Figure 3.

4. For each PE($i$,0) that is marked, broadcast the new number just received to PE($*, i \mod \sqrt{n}$). This can be done in one broadcast as follows. We first configure a link between PE($i$,0) and PE($i$, $i \mod \sqrt{n}$) and another link between PE($i$, $i \mod \sqrt{n}$) and PE(0, $i \mod \sqrt{n}$), then send the data via the configured subbus.

Thus, Nonrecursive_Sort($S$) requires 4 broadcasts $(1+1+1+1)$.

Lemma 4 Given a set $S$ of $n'$ elements allocated in the processors of the northernmost row of an $n \times n'$ R-mesh $M$. $S$ can be sorted and stored in the processors of the northernmost row of $M$ in $g(n')$ broadcasts, where

$$g(n') = \begin{cases} 16 & \text{if } \sqrt{n} \leq n' \leq \sqrt{n}^3 \\ 42 & \text{if } \sqrt{n}^3 < n' \end{cases}$$

Proof

Recursive_Sort($S$)

1. Broadcast elements of $S$ by column and row buses such that PE($*, j$) has $a_j$ and PE($i$, *) has $a_i$. This is done in one broadcast, as illustrated in Figure 4.

2. Partition $M$ into $p$ submeshes, $M_l, 0 \leq l \leq p - 1$, each of size $n \times f(n')$. For each submesh $M_l$, sort $W_l$ on $M_l$ recursively, then mark $q$ elements that evenly partition the elements allocated in the processors of northernmost row of $M_l$. This step only requires $g(f(n')) - 1$ broadcasts since the initial broadcast in the recursive call is already done in the earlier step.
3. Now that $S'$ is selected, $S$ is to be partitioned by elements in $S'$ into $m$ subsets $S_k$, $0 \leq k \leq m - 1$. Furthermore, $S_0, S_1, \ldots, S_{m-1}$ are allocated in the sequential order in the northernmost row of $M$, one element per processor. The following substeps shows the computation in $h(n')$ broadcasts, where

$$h(n') = \begin{cases} 
8 & \text{if } \sqrt[3]{n} < n' \leq \sqrt[3]{n^3} \\
10 & \text{if } \sqrt[3]{n^3} < n'
\end{cases}$$

(a) If $\sqrt[3]{n} < n' \leq \sqrt[3]{n^3}$ then do the following:

i. Call Nonrecursive_Sort to sort $S'$ (the elements marked in the processors of northernmost row of $M$). This step only takes 3 broadcasts since the initial broadcast for Nonrecursive_Sort is already done in the earlier step.

ii. Allocate $m$ submeshes, $M_0, M_1, \ldots, M_{m-1}$ in $M$, each of the same size $\sqrt[3]{n} \times n'$.

iii. For each submesh $M_k$, broadcast $s_k$ and $s_{k+1}$ to the processors of $M_k$. This step requires two broadcasts.

iv. For each submesh $M_k$, each PE($i$, $j$) of $M_k$ has $a_j$, $s_k$ and $s_{k+1}$. Extract those $a_j$'s in the northernmost row of $M_k$ such that $s_k \leq a_j < s_{k+1}$ to the westernmost column of $M_k$. Also mark those processors in the

(b) If $\sqrt[3]{n^3} < n'$ then do the following:

i. Compute the ranks of $S'$ in $S$. This requires 7 broadcasts as shown in the following:

A. Let $s_0, s_1, \ldots, s_{m-1}$ be the elements of $S'$. Partition $M$ into $m$ submeshes, $M_0, M_1, \ldots, M_{m-1}$, each of the same size $\sqrt[3]{n} \times n'$. Broadcast these $m$ numbers to each submesh $M_k$.

B. For each submesh $M_k$, broadcast $s_k$ to all the processors of $M_k$. This step is not counted since it is already done in the last step of previous recursive call. Then a list of $n'$ bits, $b_0, b_1, \ldots, b_{n'-1}$ is computed, where $b_i = 1$ if the number in PE($i$, 0) is less than $s_k$.

C. Count the number of 1's in the list of $n'$ bits on each $M_k$. We apply the parallel prefix sum algorithm of [14, 22] for this computation. It takes $\lceil \log n'/\log \sqrt[3]{n} \rceil$ iterations and requires $6$ broadcasts.
ii. If $s_k$ has rank $\Pi(k)$, it is broadcasted to $\text{PE}(0, \Pi((k)))$. Then partition $M$ into submeshes $M_0, M_1, \ldots, M_{m-1}$, where $M_k$ is of size $n \times (\Pi((k+1)) - \Pi((k))) = n \times |S_k|$.

iii. For each submesh $M_k$, broadcast $s_k$ and $s_{k+1}$ to each processor of $M_k$. This step requires two broadcasts.

iv. For each submesh $M_k$, each $\text{PE}(i, j)$ of $M_k$ has $a_i, s_k$ and $s_{k+1}$. Extract those $a_i$'s in the westernmost column of $M_k$ such that $s_k \leq a_i < s_{k+1}$ to the northernmost row of $M_k$. Note that the size of $M_k$ is $n \times |S_k|$. Thus, the computation requires one broadcast by Lemma 2.

4. Partition $M$ into $m$ submeshes $M_0, M_1, \ldots, M_{m-1}$ such that $S_k$ is stored in the northernmost row of $n \times |S_k|$ submesh $M_k$. Sort $S_k$ on $M_k$ recursively. By Lemma 1, $|S_k| \leq f(n')$. Thus, this computation requires $g(f(n'))$ broadcasts.

We calculate the number of broadcasts in each step of $\text{Recursive\_Sort}(S)$ and conclude that $\text{Recursive\_Sort}(S)$ requires $g(n') = 1 + (g(f(n')) - 1) + h(n') + g(f(n'))$ broadcasts.

Thus, $\text{Recursive\_Sort}(S)$ requires

$$g(n') = \begin{cases} 1 + 3 + 8 + 4 = 16 & \text{if } \sqrt{n} < n' \leq \sqrt{n^3} \\ 1 + 15 + 10 + 16 = 42 & \text{if } \sqrt{n^3} < n' \end{cases}$$

Recently, Olariu and Schwing presented a deterministic sampling scheme that further reduces the number of broadcasts required for the sorting problem [22]. We tune the parameters of our algorithm and obtain the same broadcast-efficiency. The procedures and analysis are depicted as follows:

Note that Lemma 1 holds for the following parameters setup:

$$f(n') = \begin{cases} \sqrt{n^3} & \text{if } n' \leq \sqrt{n^3} \\ \sqrt{n^5} & \text{otherwise} \end{cases}$$

and $q = \lceil \sqrt{n^3} \rceil$. Then the analysis follows:

$$h(n') = \begin{cases} 8 & \text{if } \sqrt{n} < n' \leq \sqrt{n^5} \\ 17 & \text{if } \sqrt{n^5} < n' \end{cases}$$

and

$$g(n') = \begin{cases} 9 \text{ (see [22])} & \text{if } \sqrt{n} < n' \leq \sqrt{n^5} \\ 1 + 8 + 17 + 9 = 35 & \text{if } \sqrt{n^5} < n' \end{cases}$$

We summarize the results of this section as a theorem.

**Theorem 3** There exists an R-mesh algorithm that takes at most 35 bus broadcasts to sort $n$ numbers on an $n \times n$ PARBUS.

**APPENDIX B: NUMBER OF BROADCASTS IN THE RMESH MODEL**

To analyze the number of broadcasts of the new sorting algorithm in the RMESH model, we note some necessary changes to the PARBUS implementation of Section 5.

- Broadcasting elements of $S$ by column and row buses such that $\text{PE}(*)(i)$ has $a_i$ and $\text{PE}(i, *)$ has $a_i$ requires only one broadcast in the PARBUS. This step is simulated by two broadcasts in the RMESH, one uses column buses, the other uses row buses, as illustrated in Figure 5.

- The connections $\{\text{WN, ES}\}$ and $\{\text{WS, EN}\}$ in the PARBUS model can be simulated using 4 PEs in the RMESH model [6]. The broadcast initiated after these types of configuration can be simulated by two broadcasts, one for the processors indexed with odd numbers, the other with even numbers.

![Row buses (left) and column buses (right) in the RMESH.](image)
• Counting the number of 1’s in the list of \( n' \) bits on an \( \sqrt{n} \times n' \) RMESH.

Consider \( n' \leq \sqrt{n} \). The implementation is similar to that of Nonrecursive_Sort. It takes 2 iterations (one for odd bits, and the other for even bits) with each iteration using 2 broadcasts (one for counting bits, and the other for broadcasting to the top left corner of the mesh).

Consider \( n' > \sqrt{n} \). We use only 12 broadcasts on an \( \sqrt{n} \times n' \) PARBUS. But the PARBUS implementation also utilizes the connection \{WE,NS\}. The addition algorithm in [9] is adopted for the computation in the RMESH model. Lemma 5 shows that it requires 19 broadcasts.

**Lemma 5** There exists an implementation that counts the number of 1’s in the list of \( n' \) bits \((n' > \sqrt{n})\) on an \( \sqrt{n} \times n' \) RMESH in 18 broadcasts.

**Proof**

1. Partition \( M \) into \( \sqrt{n} \) submeshes, \( M_0, M_1, \ldots, M_{\sqrt{n}-1} \), each submesh is of the same size \( \sqrt{n} \times n' / \sqrt{n} \).
2. For each submesh \( M_k \), count the number of 1’s in the list of \( n'/\sqrt{n} \) bits on each \( M_k \). This requires 4 broadcasts since \( n'/\sqrt{n} \leq \sqrt{n} \).
3. Add the numbers calculated in \( M_k \) and \( M_{k+1} \) for \( k \) that is even. This requires one broadcast. Add the numbers calculated in \( M_k \) and \( M_{k+2} \) for \( k \) such that \( k \mod 4 = 0 \). This requires another broadcast. Let these \( \sqrt{n}/4 \) numbers be \( x_0, x_1, \ldots, x_{\sqrt{n}/4-1} \).
4. Partition \( M \) into 2 \( \log n \) submeshes \( M_1, M_2, \ldots, M_{2\log n} \) each of the same size \( \sqrt{n} \times \sqrt{n} \).
5. Compute the sum of these \( \sqrt{n}/4 \log n \) bits numbers, \( x_0, x_1, \ldots, x_{\sqrt{n}/4-1} \). Stores the \( k \)th bit of the sum in \( M_k \). This requires 9 broadcasts, as shown in the following steps.

(a) Distribute these \( \sqrt{n}/4 \) numbers to each \( M_k \) such that PE(0, \( 4\ast i \)) of \( M_k \) stores the \( k \)th bit of \( x_i \). This require 3 broadcasts, one for collecting \( x_i \)'s to the west most column, and two for distributing \( x_i \)'s to each \( M_k \).

(b) Apply the Carry Lookahead generation circuit in [9] to sum up these \( \sqrt{n}/4 \log n \) bits numbers. This require 6 broadcasts (three for computing bit sums of \( x_i \)'s, three for computing carry lookahead).

6. Convert binary bits of the sum to a number and store it in the register of top left corner of \( M \). This requires 3 broadcasts, as shown in the following step.

(a) Collect these bits to the west most column. (This requires 1 broadcast.)
(b) Store the \( i \)th binary bit of \( j \) in PE(\( i,j \)) for each \( i \) and \( j \).
(c) Broadcast binary bits of the sum from the westernmost column by row buses. (This requires 1 broadcast.)
(d) Mark PE(\( i,j \)) if its store bit and received bit from the broadcast matches for each \( i \) and \( j \).
(e) Set the interconnection pattern of PE(\( i,j \)):

\[
A(i,j) = \begin{cases} 
\{SNWS\} & \text{if PE}(i,j) \text{ marked} \\
\{WS\} & \text{if PE}(i,j) \text{ unmarked and } i = 0 \\
\{\} & \text{otherwise} 
\end{cases}
\]

(f) Broadcast \( j \) from PE(\( \sqrt{n},j \)) for each \( j \). (This requires 1 broadcast.)

Thus, this requires \( 18(=4 + 2 + 9 + 3) \) broadcasts.

Now we calculate \( h(n') \) in the RMESH model and obtain:

\[
h(n') = \begin{cases} 
13 & \text{if } \sqrt{n} < n' \leq \sqrt{n}^3 \\
23 & \text{if } \sqrt{n}^3 < n'
\end{cases}
\]

Thus, \( g(n') \) in the RMESH model is:

\[
g(n') = \begin{cases} 
1 + 1 + 2 + 2 = 6 & \text{if } n' \leq \sqrt{n} \\
2 + 5 + 13 + 6 = 26 & \text{if } \sqrt{n} < n' \sqrt{n}^3 \\
2 + 24 + 23 + 26 = 75 & \text{if } \sqrt{n}^3 < n'
\end{cases}
\]

**Theorem 4** There exists an \( R \)-mesh algorithm that takes at most 75 bus broadcasts to sort \( n \) numbers on an \( n \times n \) RMESH.
APPENDIX C: A SCALABLE SORTING ALGORITHM

The algorithm is a recursive one, so we describe it as one that sorts \( n' \) elements on an \( n/K \times n/K \) R-mesh. \( T(n') \) denotes the time complexity of our algorithm in sorting \( n' \) elements on an \( n/K \times n/K \) R-mesh.

New Scalable Sorting Algorithm

**Input:** a set \( S \) of \( n' \) elements \( a_0, a_1, \ldots, a_{n'-1} \) allocated in the processors of northernmost rows of \( M \). Specifically, \( a_i \) is allocated in its original position – \( PE(\lfloor i/(n/K) \rfloor, \mod(n/K)) \).

**Output:** \( a_i \) is allocated in its sorted position – \( PE(\lfloor i/(n/K) \rfloor, \mod(n/K)) \) if the rank of \( a_i \) in \( S \) is \( \Pi(i) \).

**Auxiliary Variables:** \( q = \lceil \sqrt{K} \rceil, p = [n'/\lceil f(n') \rceil], \) and \( m = p \times q \).

1. If \( n' \leq n/K \) then sort by the method in Section 4. (It takes \( O(1) \) time.)
2. Select a set \( S' \) of \( m+1 \) elements \( s_0, s_1, \ldots, s_m \) from \( S \) such that the number of elements \( x \) in \( S \) satisfying \( s_k \leq x < s_{k+1} \) is at most \( f(n') \). Also store elements of \( S' \) in the northernmost row of \( M \). See procedure \( \text{Select}(S) \) for details. (This step requires \( D(n') \) time.)
3. Sort \( S' \) on \( M \). Let \( s_0, s_1, \ldots, s_{m-1} \) be the sorted list. This step requires \( O(1) \) time since \( m \leq K \leq n/K \).
4. Partition \( S \) into \( m \) subsets \( S_k, 0 \leq k \leq m-1 \) such that \( S_k = \{ s \in S | s_k \leq s < s_{k+1} \} \). Sort \( S_k \) and store elements of \( S_k \) at their sorted positions in \( M \). These are implemented by doing the following steps sequentially for \( 0 \leq k \leq m-1 \):
   (a) Elements of \( S_k \) are routed to the northernmost rows of \( M \). (See procedure \( \text{Route}_1(S_k) \). This step requires \( R_1(\lvert S_k \rvert) \) time.)
   (b) Sort \( S_k \) on \( M \) recursively using this algorithm. (This step needs \( T(\lvert S_k \rvert) \) time.)
   (c) Route the elements of sorted \( S_k \) to their sorted positions. (See procedure \( \text{Route}_2(S_k) \). This step requires \( R_2(\lvert S_k \rvert) \) time.)
5. Combine the sorted subsets into a single sorted list. (It takes \( C(n') = O(1) \) time.)

These \( s_k \)’s can be located by the following procedure. The proof that these elements selected do have the property as required by the scalable sorting algorithm, is similar to that of Lemma 1. We omit it for brevity.

**Select(S)**

1. Evenly divide \( S \) into \( p \) subsets, \( W_l, 0 \leq l \leq p-1 \), where the elements in \( W_l \) are not necessarily smaller than those in \( W_{l+1} \) and so this step is trivial. Note that \( \lvert W_l \rvert = f(n') \) for \( 0 \leq l \leq p-2 \), and \( \lvert W_p \rvert \leq f(n') \) for \( l = p-1 \). Also note that \( p \leq \sqrt{K} \).
2. Do the following steps sequentially for \( 0 \leq l \leq p-1 \):
   (a) Sort \( W_l \) on \( M \). Note that additional time is needed to route \( W_l \) to the northernmost rows of \( M \). (This can be easily done in \( O(f(n')(n/K)) \) time.)
   (b) From the sorted \( W_l \) select \( w_{l,0}, w_{l,1}, \ldots, w_{l,q-1} \) such that these elements evenly partition \( W_l \) into subsets of size \( f(n')/q \). (This step obviously takes \( O(1) \) time.)
   (c) Route the set of all the elements selected in the preceding step to the northernmost row of \( M \), in fact, \( w_{l,l'} \) is routed to \( PE(0, l \times q + l') \). This can be easily done in \( O(1) \) time since the locations that \( w_{l,l'} \)'s were routed to are known.

   (This requires \( p \times (T(f(n')) + O(f(n')(n/K))) \) time.)
3. Let \( S' = \{ s_k : 0 \leq k \leq m-1 \} \) be the set of all the elements selected in the preceding step. Note that \( s_m = \infty \) is included in \( S' \) for convenience of presentation.

Now that each \( W_l \) is already sorted and then stored in a dedicated submesh \( M_l \) of size \( n/K \times f(n')/(n/K) \) after running the procedure \( \text{Select}(S) \). We are ready to describe the following
two routing procedures. We first define some notations relative to $S_k$. Suppose $a \in S_k$ is marked and currently stored in $PE(i,j)$, then $pos(a) = (i,j)$. Define the row major order of $a$ in $S_k$ as $\{(a' \in S_k : pos(a') < pos(a) \text{ (lexicographically))}\}$.

**Route_1($S_k$)**

1. Mark $S_k$ (the elements in $S$ that are between $s_k$ and $s_{k+1}$). This is implemented by marking the elements in $W_l$ that are between $s_k$ and $s_{k+1}$ on $M_l$ for each $l$ in parallel. (Note that $1 \leq l \leq p \leq \sqrt{K}$)
2. Route these $|S_k|$ marked numbers to northernmost rows of $M$. This step includes the following substeps.
   (a) Compute the row major order of each element of $S_k$ as follows.
      i. Count the number of marks on $M_i$ by bus splitting for each submesh of $M_i$ in parallel. This step requires $O(1)$ time since $W_1$ is already sorted.
      ii. Compute the prefix sum of these $p$ numbers obtained in the previous step. This requires $O(1)$ time since $p < n'$. 
   (b) Route $n/K$ of these marked elements sequentially to northernmost rows until all marked elements are routed. This can be easily done since the row major orders of these marked numbers are computed. This step takes $K \times |S_k|/n$ time.

**Route_2($S_k$)**

1. Compute the ranks in $S$ of elements of $S_k$. This can be easily done since $S_k$ is sorted and elements of $S_{k-1}$ are stored in their sorted positions.
2. Route $n/K$ elements of $S_k$ sequentially to their sorted positions. This can be easily done since the ranks in $S$ of elements of $S_k$ is computed. This step takes $K \times |S_k|/n$ time.

We now analyze the time complexity of our new scalable sorting algorithm.

$$f(n') = \begin{cases} n/\sqrt{K} & \text{if } n/\sqrt{K} < n' \\ n/K & \text{otherwise} \end{cases}$$

$$m = \lceil n'/\lfloor f(n') \rfloor \rceil \times \lceil \sqrt{K} \rceil.$$ 

$$T(n') = \begin{cases} D(n') + \sum_{k=0}^{m-1} R_1(|S_k|) + T(|S_k|) & \text{if } n/K < n' \\ O(1) & \text{otherwise} \end{cases}$$

$$D(n') = \begin{cases} O(1) & \text{if } n/K < n' \\ \frac{n' / \lceil f(n') \rceil} {m} \times T(f(n')) + O(f(n') / (n/K)) & \text{otherwise} \end{cases}$$

We use the amortization technique [4] to calculating the time complexity of routing procedures Route_1 and Route_2.

$$\sum_{k=0}^{m-1} R_1(|S_k|) + R_2(|S_k|) = \sum_{k=0}^{m-1} 2 \times K \times |S_k| / n$$

$$= O \left( m + 2 \times K \times n \sum_{k=0}^{m-1} |S_k| \right)$$

$$= O(m + 2 \times K \times n \times n').$$

Solving the recurrence equations yields

$$T(n') = \begin{cases} O(n'/\sqrt{K}) & \text{if } n/\sqrt{K} \leq n' \leq n \\ O(n'/n \times \sqrt{K^3}) & \text{if } n/K < n' \leq n/\sqrt{K} \text{ otherwise} \end{cases}$$

Thus, we have

$$T(n) = O(K).$$

**Authors’ Biographies**

Ten H. Lai is on the faculty of Computer and Information Science at the Ohio State University.
He has published in *IEEE Transactions on Computers, SIAM Journal on Computing, Journal of Algorithms, Journal of Parallel and Distributed Computing*, on topics ranging from graph embedding, task scheduling, subcube management strategies for hypercubes, parallel algorithms, distributed algorithms, to VLSI layout design. His current research interests are in the areas of parallel and distributed computing and communication protocols. Dr. Lai is a member of the Association for Computing Machinery and the IEEE Computer Society. He served on the Editorial Board of the *IEEE Transactions on Parallel and Distributed Systems* from January 1993 to December 1996.

**Ming-Jye Sheng** received his Ph.D. degree in computer science from the Ohio State University in 1994. He has been involved in telecommunication projects with AT & T during 1994–1995. He also served as the director of R & D of CyberExpress Inc. and initiated several internet projects during 1995–1996. He is currently a member of technical staff of Lucent Technologies. His current research and development interests are new technology and performance analysis of wireless systems.
