VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Sparse Token Distribution on Reconfigurable Meshes

Heiko Schröder and **Gavin Turner**

Technical Report CS-TR-96/24
November 1996

Abstract

A solution to the k-sparse token distribution problem is presented for the 2-dimensional reconfigurable mesh. The algorithm, being the first algorithm proposed to solve this problem under any model of reconfiguration, is shown to be asymptotically worst-case optimal in reducing the discrepancy between maximum and minimum processor loads by 1 in $\Theta(\sqrt{k})$ time steps. In addition, the algorithm provides an optimal solution to more general (non-sparse) token distribution problems.

Author Information

Heiko Schröder studied mathematics, physics, and computer science at CAU in Kiel (Germany) where he received his Ph.D. in computer science in 1977. Until 1989 he was Assistant Professor of Computer Science at CAU and Kansas University (U.S.A.) and Senior Research Fellow at ANU in Canberra (Australia). Until 1994 he was Professor of Microelectronics in Newcastle (Australia) and is currently Professor of CS in Loughborough (U.K.). His main research area is massively parallel computing including algorithms and architectures for problems related to sorting, image processing and fault tolerance.

Gavin Turner received a Ph.D. in Computer Science at the University of Newcastle, Australia in 1996, and is currently a lecturer in Computer Science at Victoria University of Wellington, Wellington New Zealand. His main research interests lie in the area of design and analysis of algorithms, in particular algorithms and architectures for special-purpose parallel computing. In addition, he is interested in a wide variety of problems from image processing, load balancing and combinatorics.

Sparse Token Distribution on Reconfigurable Meshes

Heiko Schröder
H.Schroder@lut.ac.uk
Department of Computer Studies,
University of Loughborough, LE11 3TU, UK

Gavin Turner
gavin@comp.vuw.ac.nz
Department of Computer Science,
Victoria University of Wellington, PO Box 600, Wellington, New Zealand

Abstract

A solution to the k-sparse token distribution problem is presented for the 2-dimensional reconfigurable mesh. The algorithm, being the first algorithm proposed to solve this problem under any model of reconfiguration, is shown to be asymptotically worst-case optimal in reducing the discrepancy between maximum and minimum processor loads by 1 in $\Theta(\sqrt{k})$ *time steps. In addition, the algorithm provides an optimal solution to more general (non-sparse) token distribution problems.*

Keywords : Sparse Loads, Token Distribution, Reconfigurable Meshes

1 Introduction

One of the fundamental data distribution problems on parallel architectures is that of *token distribution*, a static variant of the well-studied load balancing problem, and akin to the problems of sorting and permutation routing. Each processing element (PE) of the parallel architecture possesses an initial set of *tokens*, each of which represents a task to be performed; the number of tokens stored at a particular PE is called the *load* of that PE. Ideally, one would prefer that the distribution of the tokens over the set of PEs be as even as possible, as imbalances would result in a delay in the time needed to perform all the tasks. The goal of a token distribution algorithm is to redistribute the tokens in such a way that the final loads of the PEs differ by as little as possible. Here it is assumed that each token requires only a constant amount of time to send from one PE to an adjacent PE, and that no tokens are created or destroyed before the redistribution is complete.

One of the simplest and therefore one of the most practical fixed connection networks is the mesh. In this model, the processing elements of the parallel machine are arranged in a square grid, and are connected to their neighbours by short, fixed, uni-directional links. A dynamic variant of the fixed connection mesh, the so-called *reconfigurable mesh*, has become popular with researchers over the last few years [6]. Under this model, which has been shown to have hardware cost comparable with that of the fixed connection mesh [7], the PEs are augmented with a small switching network with which they may locally short connections to form busses. The popularity of the reconfigurable mesh model is due to the fact that, while it shares many of the advantages of the fixed-connection mesh – it is extremely suitable for VLSI fabrication and therefore massively parallel computing – it removes one of the meshes greatest drawbacks – its large diameter. The reconfigurable mesh has an effective diameter in $\mathcal{O}(1)$.

Many solutions to the token distribution problem have been proposed for a variety of interconnection networks, including reconfigurable meshes. A comprehensive survey of this work may be found in [1]. In this paper, we consider an important variation to the token distribution problem - the so-called *sparse* token-distribution problem. In this problem, we assume that the distribution is such that only a small, possibly constant number of PEs contain the maximum and minimum number of tokens, reducing the time required (in theory) to reduce the overall discrepancy of the token distribution (the difference between the maximum number of tokens and minimum number of tokens in any PE).

On the conventional fixed-connection mesh, the performance of any solution to the sparse token-distribution problem is constrained by the diameter of the network. Since the diameter and the bisection width of the fixed-connection mesh ultimately reveal the same lower-bound, that is $\Omega(n)$ for an $n \times n$ mesh, a solution to the general token distribution problem (one in which no assumptions are made about the distribution) also provides an optimal solutions to sparse problems. On the reconfigurable mesh however, with its diameter $\in \mathcal{O}(1)$, there is significant room for improvement. If we have only k PEs with the maximum and minimum number of tokens, then a bisection lower bound of $\Omega(\sqrt{k})$ applies, as opposed to $\Omega(n)$ for the general case [1] (which corresponds to the case where $k \in \mathcal{O}(n^2)$).

To date, no algorithms have been proposed to solve the sparse token distribution problem on reconfigurable architectures. Many algorithms exist for solving the general token distribution problem [5, 2], but all realise an upper bound of only $\mathcal{O}(n)$ for both general and sparse distributions. In this paper we propose an algorithm which solves sparse token distribution problems on the reconfigurable mesh. This solution is optimal with respect to the bisection bound mentioned above. In addition, the algorithm provides an optimal solution to the general token distribution problem (where $k \in \mathcal{O}(n^2)$)

The organization of the paper is as follows: in the next section, we describe the model of computation. In Section 3, we formalise the sparse token distribution problem and prove a lower bound on its complexity for 2-dimensional mesh-connected architectures. The statement and analysis of the algorithm appears in Sections 4 and 5. Concluding remarks are made in Section 6.

2 Model of Computation

We denote by M_n the $n \times n$ reconfigurable, 2-dimensional mesh (*r-mesh*) with side length n. We assume that each dimension (side) of the mesh is indexed $0, \cdots, n-1$. A processor is identified as $P_{i,j}$ with i being the row index, and j the column index. Each processor (except for those at the boundary) has 4 neighbouring processors, and the processors are connected via a reconfigurable bus system consisting of a fixed connection network (FNET) and n^2 identical switching networks (RNETs) (one per processor).

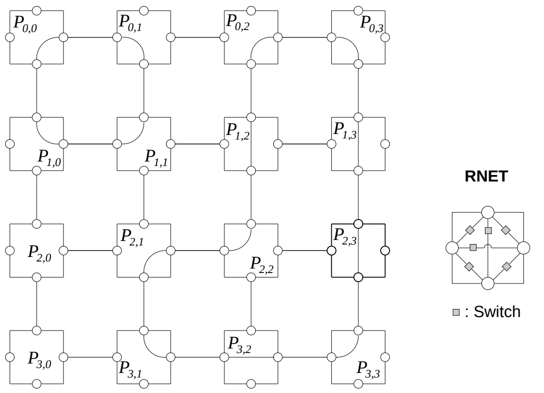

Figure 1: Pictorial Representation of the 2-dimensional Reconfigurable Mesh

Each processor has 4 *ports*. The FNET connects each processor to its 4 neighbouring processors by connecting pairs of ports. The standard 4 neighbourhood notation N,S,E,W is used to denote individual ports. The array of processors together with the FNET is identical to a SIMD fixed connection mesh. An RNET is a switching network present inside each processor which is connected to the 4 FNET ports of the processor. The processor can dynamically configure this switching network so as to electrically short the incoming ports, meaning that any incoming signal on one port will be transmitted to the ports with which it is shorted. Each processor can locally control the bus system using these switches and can write onto or read from any one of the ports.

The configuration of the RNET within each processor will be specified by grouping together ports which have been shorted. Thus if in a processor all ports are electrically isolated, its RNET configuration is denoted as (N,S,E,W). Similarly if only the North and South ports are shorted, the RNET is said to be configured (NS,E,W). A reconfigurable mesh might support all possible shortings. Some examples of these shortings along with their pictorial representation are shown in Figure 2.

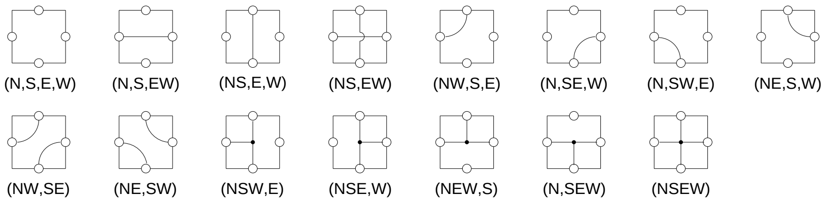

Figure 2: Examples of possible 2-Dimensional switch configurations, and their pictorial representation

We assume an SIMD architecture where in each atomic instruction a PE:

a) *configures its local RNET,*

b) *writes data to any one port register,*

c) *reads from any one register, including I/O (port) registers,*

d) *performs an elementary calculation,*

e) *writes data to an internal register.*

For each of its 4 ports, a PE contains a dedicated I/O register. It is assumed that the result of the write phase is that the contents of the respective I/O register of the writing PE is replicated in all I/O registers connected to the bus. We assume CREW buses which means that if at any time 2 PEs attempt to write to the same length of bus, the contents of all I/O registers connected to that bus will be corrupted and an error flag will signal. We make no restrictions on the way in which PEs may configure their local switching network to form busses, or on the way they read from or write to local ports. It is the responsibility of any algorithm to ensure that CREW bus access is observed.

3 The Sparse Token Distribution Problem

The sparse token distribution problem *STD*$(A;\Delta,M,k)$, being an interesting variation of the general token distribution originally proposed by Peleg and Upfal in [8], can be stated as follows: Given parallel architecture A containing n^2 processors $P_1,...,P_{n^2}$, with each processor P_i containing a stack of $\mu \leq l(P_i) \leq M$ tokens (for all $1 \leq i \leq n^2$), and for a global discrepancy between loads equal to $\Delta = M - \mu > 1$, distribute the tokens such that at the end the global discrepancy has been reduced by 1, that is, each PE with M tokens has lost a token, or all PEs with μ tokens has gained a token.

In addition, the distribution of PEs with M and μ tokens respectively is assumed to be *sparse*. Specifically, the total number of these PEs is assumed to be less than some constant $k \in \mathcal{O}(1)$. In particular, such a token distribution is said to be *k-sparse*. For any 2-dimensional mesh connected architecture, a lower bound on the time required to solve sparse token distribution problem *STD*$(A;\Delta,M,k)$ is easily derived. The result is stated in Lemma 1.

Lemma 1 *Let A be an n-by-n mesh-connected architecture with diameter d. Any solution to problem* STD$(A;\Delta,M,k)$ *requires time in* $\Omega(\sqrt{k}+d)$.

Proof To reduce $\Delta = 2$ to 1, tokens must be moved from maxima to minima such that either all maxima loose a token or all minima gain a token. In the worst-case, a set of $k/2$ maxima can be packed into a square area of side-length $\lceil\sqrt{k/2}\rceil$. A set of $k/2$ minima can be packed into a similar area. In the case of $\Delta = 2$, tokens must be moved from maxima to minima until one set has been removed. To move $\sqrt{k/2}$ tokens out of a square area of side-length $\lceil\sqrt{k/2}\rceil$ requires time proportional to $\sqrt{k}$. Similarly to move $\sqrt{k/2}$ tokens into a square area of side-length $\lceil\sqrt{k/2}\rceil$ requires time proportional to $\sqrt{k}$. The maximum distance *between* two such areas is the diameter d of the network. Therefore, in the case where $\sqrt{k} \in O(d)$ an algorithm to solve problem $STD(A; \Delta, M, k)$ requires time in $\Omega(d)$. Similarly, in the case where $d \in O(\sqrt{k})$, an algorithm to solve problem $STD(A; \Delta, M, k)$ requires time in $\Omega(\sqrt{k})$. Therefore the lemma holds. □

4 Sparse Token Distribution Algorithm

In this section we present our algorithm, SparseBalance, which reduces the discrepancy Δ of a k-sparse token distribution by 1 on the reconfigurable mesh in at most $\mathcal{O}(\sqrt{k})$ token transfer steps. The algorithm is based on the ability to pack k data elements from arbitrary (but distinct) mesh positions into a $\lceil\sqrt{k}\rceil \times \lceil\sqrt{k}\rceil$ square area of PEs. In addition, this ability must be *reversible* - it must be possible to return each data element to its PE of origin (given that each data element carries with it its starting address). Assuming that a method to implement such a pack exists, algorithm SparseBalance is stated in Figure 3. Informally, the algorithm operates as follows.

1 **Initialisation**

- 1.1 Minima at mesh location (i, j) generate an *address packet* containing (i, j) and maxima at location (a, b) generate a *token packet*, containing (a, b) and a single token. (These are collectively referred to as the *elements*)

2 **Pack and Match Tokens with Addresses**

- 2.1 Using a k-element packing algorithm, pack all *elements* into a $\lceil\sqrt{k}\rceil \times \lceil\sqrt{k}\rceil$ set of PEs (a *bucket*) in any order.
- 2.2 Each processor containing an element appends it's unique address to the it (denoted the *temporary* address).
- 2.3 Sort or pack the token packets in the bucket into the upper-left positions of the bucket (ie. row-major order).
- 2.4 Likewise, sort or pack the address packets.
- 2.5 Each bucket processor containing both a token and an address packet attaches the token to the address packet (the token has been matched) and discards the token packet. Any unmatched address packets are also discarded.
- 2.6 All remaining packets are routed back to the temporary address contained in their packet.

3 **Route Matched Tokens to Minima**

- 3.1 Using the reverse of the k-element packing algorithm, unpack all remaining packets such that each packet is routed the mesh address stored in the packet.

Figure 3: Algorithm SparseBalance

To remove either all maxima (each maxima looses a token) or all minima (each minima gains a token), a matching process is undertaken whereby tokens from maxima are matched with the addresses of minima. This matching process is completed as follows. A single token is chosen from each maxima, and for each minima a packet is generated containing the minima's mesh address (Figure 4 pane 1). The set of k such tokens and addresses are packed into a $\lceil\sqrt{k}\rceil \times \lceil\sqrt{k}\rceil$ block of processors (this square area is denoted the *bucket*) (Figure 4 pane 2). Within the bucket, single tokens are matched to unique address packets (Figure 4 pane 3). Matched tokens are then routed to the corresponding minima (the minima identified in the matched address), and unmatched tokens (if they exist) are routed back to their PE of origin (Figure 4 pane 4). The result is that either all tokens have been matched (all maxima loose a token) or all addresses have been matched (all minima receive a token) and therefore that Δ has been reduced by 1. The algorithm is described in more detail in the following sections.

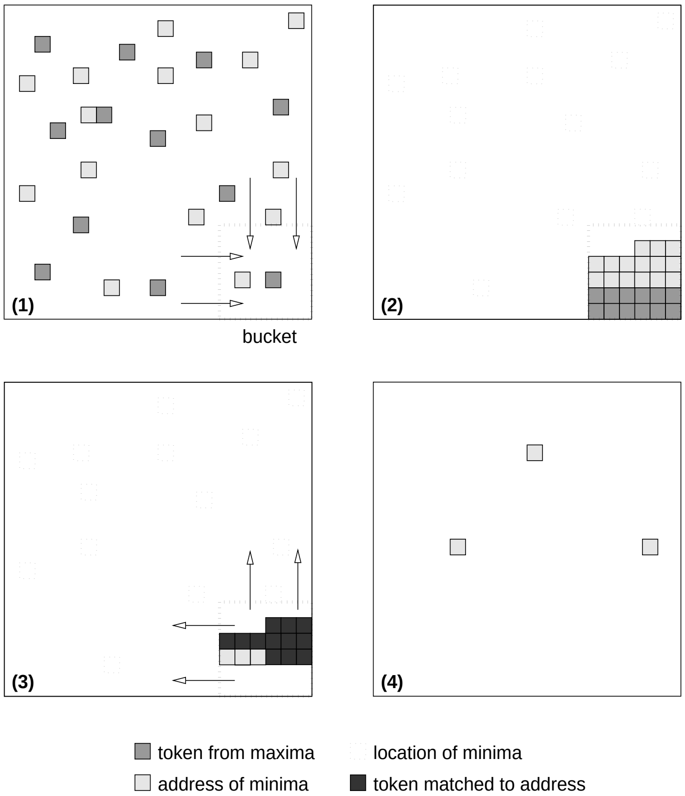

Figure 4: Pictorial Overview of Algorithm SparseBalance

4.1 Phase 1: Initialisation

The first phase of the algorithm (step 1.1) is a constant time operation which simply serves to initialise the data packets (herein collectively denoted *the elements*) which will be moved across the network in subsequent phases. Each maxima and minima create a data packet with structure shown in Figure 5 below. The first field of the packet, having size 1-bit, indicates if it has been created by a maxima (type 0) or a minima (type 1). The second field, of size $\log N$ bits, stores the mesh address of the PE which created it. The third field of the packet is used as temporary storage and has size $\log N$ bits. When creating their packet, both maxima and minima leave this field blank. The final field, having size T bits, is used to store a single token. All maxima choose a token from their local stack and place it into this field. Conversely, all minima leave this field blank. Packets of type 0 will be denoted *token packets*, and packets of type 1 will be denoted *address packets*. Collectively, the set of packets will be simply denoted the *elements*.

4.2 Phase 2: Pack and Match Tokens with Addresses

In phase 2 of the algorithm, each token packet is matched with an address packet, or each address packet is matched with a token packet, whichever is the lesser. This is achieved with the help of a non-trivial

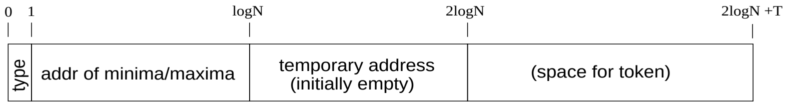

Figure 5: Structure of the Address Packet Constructed by Each Maxima and Minima

k-element packing algorithm which shall be described in the next section. Such a packing algorithm takes k data elements which are arbitrarily distributed throughout the mesh, and moves each of them into an arbitrary (but unique) location within a square $\lceil\sqrt{k}\rceil \times \lceil\sqrt{k}\rceil$ area. The algorithm proposed in this paper completes this process in optimal $\mathcal{O}(\sqrt{k})$ steps.

The matching process is completed as follows. Initially, we pack all k elements into such a $\lceil\sqrt{k}\rceil \times \lceil\sqrt{k}\rceil$ square area of processors (denoted the *bucket*). Each bucket PE which contains an element stores its own mesh address in the second field (the temporary storage area) of the packet. Remembering an elements initial bucket location allows us to return the element to this location - a process which will be useful for Phase 3.

The token packets, and then the address packets are individually sorted into row major order (within the bucket) such that the first $\min(|maxima|, |minima|)$ bucket PEs hold packets of each type. This can be achieved using any of several reconfigurable-mesh sorting algorithms [3, 4] in time $\mathcal{O}(\sqrt{k})$. These PEs take their token from the token packet (the packet is then discarded) and store it in the fourth field of the address packet - the token has been matched to a minima (and this minima will be its final destination). PEs which hold only an address packet discard it.

The remaining packets, now only one per PE and each containing a token, are routed back to the temporary address (within the bucket) stored in the third field. Again, this may be achieved using one of several reconfigurable mesh routing algorithms [3, 4]. All that then remains is to return each of the remaining packets to their original mesh location (the PE which created them). This is achieved in the third and final phase of the algorithm.

4.3 Phase 3: Route Matched Tokens to Minima

The third phase simply routes all matched tokens to their respective minima. This can be achieved in time $\mathcal{O}(\sqrt{k})$ by reversing the k-element packing algorithm. Note that the packing algorithm presented here in the next section may be easily reversed to give such a result.

4.4 Analysis

Given that we can pack and unpack k data elements into a $\lceil\sqrt{k}\rceil \times \lceil\sqrt{k}\rceil$ square area of processors in $\mathcal{O}(\sqrt{K})$, the asymptotic complexity of the algorithm can be easily determined.

- Phase 1: initialisation $\in \mathcal{O}(1)$
- Phase 2: pack $\in \mathcal{O}(\sqrt{k})$ + sort $\in \mathcal{O}(\sqrt{k})$ + route $\in \mathcal{O}(\sqrt{k})$
- Phase 3: reverse-pack $\in \mathcal{O}(\sqrt{k})$

Therefore our algorithm is in $\mathcal{O}(\sqrt{k})$ and therefore worst-case optimal with respect to our bisection bound.

5 An Efficient k-Element Packing Algorithm

In this section, we present an $\mathcal{O}(\lceil\sqrt{k}\rceil)$ algorithm which packs an arbitrarily distributed set of k data items (herein denoted *elements*) into a $\lceil\sqrt{k}\rceil \times \lceil\sqrt{k}\rceil$ square area of PEs (herein denoted the *bucket*). In stating the algorithm we arbitrarily choose the bucket to be the PEs in the bottom right of the mesh (those PEs (i, j) where i and $j \geq n - \lceil\sqrt{k}\rceil$). The algorithm may be trivially altered for any bucket position.

The algorithm is formally stated in Figure 6. Informally, the algorithm operates as follows. The algorithm is based on a repeating dual-phase process, using alternate horizontal (row-wise) and vertical (column-wise) phases, to fill the bucket with elements based on a row-major ordering of its PEs.

1 **Initialisation**

1.1 Processors in the top row of the bucket, $\mathrm{PE}_{i,(n-\lceil\sqrt{k}\rceil+1)}$, $(n - \lceil\sqrt{k}\rceil + 1) \leq i < n$, are members of set `read`.

1.2 Processor $\mathrm{PE}_{(n-\lceil\sqrt{k}\rceil+1),(n-\lceil\sqrt{k}\rceil+1)}$ is labeled the `start` processor.

2 **Repeat for** $2 \times \lceil\sqrt{k}\rceil$ **Iterations**

2.1 **Horizontal Step**

2.1.1 All PEs containing elements which have not yet been placed in the bucket cut the row ring bus and send their element to the East. All PEs which exist in the same bucket column as members of the `read` set attempt to read an element from the West.

2.1.2 All PEs which received a value in step 2.1.1 are marked, and configure (NE, SW), except those in the same column as `start` which configure (NE, S, W). All others configure (NS, EW). PE `start` writes a binary signal # to the South. All marked processors attempt to detect signal # from the North.

2.1.3 All PEs configure (NS, EW), except those marked PEs which detected signal # in step 2.1.2 who configure (N, E, SW) (linking a row bus to a column bus). All PEs containing elements which have not yet been placed in the target area cut their row bus.

2.1.4 All PEs containing elements which have not yet been placed in the target area write their element to the East, and all attempt to read an element from the West. All PEs in set `read`, attempt to read an element from the North.

2.1.5 All non-bucket PEs configure (NS, EW). All bucket PEs configure (N, S, E, W) except the first column of bucket PEs which configures (NE, SW). All bucket PEs which received an element in step 2.1.4 write a 1 to the East. All other bucket PEs write a 0 to the East. All PEs read a binary signal from the left along their *stepped bus* (see figure).

2.1.6 The single PE which wrote a 0 and received a 1 becomes the `start` PE.

2.1.7 All PEs set (NS, EW) and the `start` PE broadcasts its index up the column bus. All PEs attempt to read this index and determine if they are in the bucket's `start` column.

2.1.8 All PEs configure (N, S, E, W). All bucket PEs which received an element in step 2.1.4 write a 1 to the South. All other bucket PEs write a 0 to the South. All bucket PEs attempt to detect a 1 to the North.

2.1.9 All PEs which detected a 1 in step 2.1.8 become members of set `read`. All those which sent a 1 in step 2.1.8 are removed from set `read`. Read set members broadcast a signal in their column to indicate their membership.

2.2 **Vertical Step**:

Repeat the horizontal process, using the diagonal to turn a vertical step into a horizontal step. That is, all PEs with elements to be packed now send the element down their column at steps 2.1.1 and 2.1.4 instead of along their rows.

Figure 6: Algorithm to solve the k-Element Packing Problem

The algorithm begins by defining a set of $\lceil\sqrt{k}\rceil$ bucket PEs, the *ready-set*, to be the set of PEs which will accept elements in the first (horizontal) data movement phase. After each subsequent data movement phase (vertical or horizontal) the ready-set is updated to be the first $\lceil\sqrt{k}\rceil$ bucket PEs (in row-major order) which have not yet received an element.

In each horizontal phase, the topmost $\lceil\sqrt{k}\rceil$ rows which contain elements (or less if fewer such rows exist) are identified. These rows are then connected to the $\lceil\sqrt{k}\rceil$ columns which contain members of the the ready-set (all $\lceil\sqrt{k}\rceil$ columns of the bucket) via the row and column busses, and a single element from each row is sent to the corresponding read-set member. The phase ends by identifying the members of a new read-set; the next $\lceil\sqrt{k}\rceil$ bucket PEs (in row-major order) which have not yet received an element.

The vertical phase is implemented in an equivalent manner to the horizontal phase. The exception is that now the first $\lceil\sqrt{k}\rceil$ columns which contain elements (or less if fewer such columns exist) are identified and are linked via column and row busses to the columns of the read-set. This is achieved using the $i = j$ diagonal to effectively turn each source column into a source row, and then using the method for horizontal transfer.

The horizontal phase of the algorithm is described in more detail in sections 5.1 and 5.2. Necessary extensions needed to implement the vertical phase are discussed in section 5.3, and the return of elements from the bucket to their PE of origin (a necessary extension to the algorithm) is discussed in section 5.5.

5.1 Moving Packets to The Read Set

The first stage of a horizontal data movement phase involves the movement of one or more elements into members of the read-set. This process begins by matching up to $\lceil\sqrt{k}\rceil$ rows which contain elements to the columns of the bucket. This is achieved in two steps. In step 2.1.1, all rows containing elements form a horizontal ring, upon which a signal is broadcast by all element owners. All PEs which reside in columns which also hold bucket PEs attempt to detect this signal on the bus (See Figure 7 pane 1). The detection of this signal identifies a row which contains elements, and PEs which have detected the signal are marked with a crossed-circle in Figure 7 pane 1.

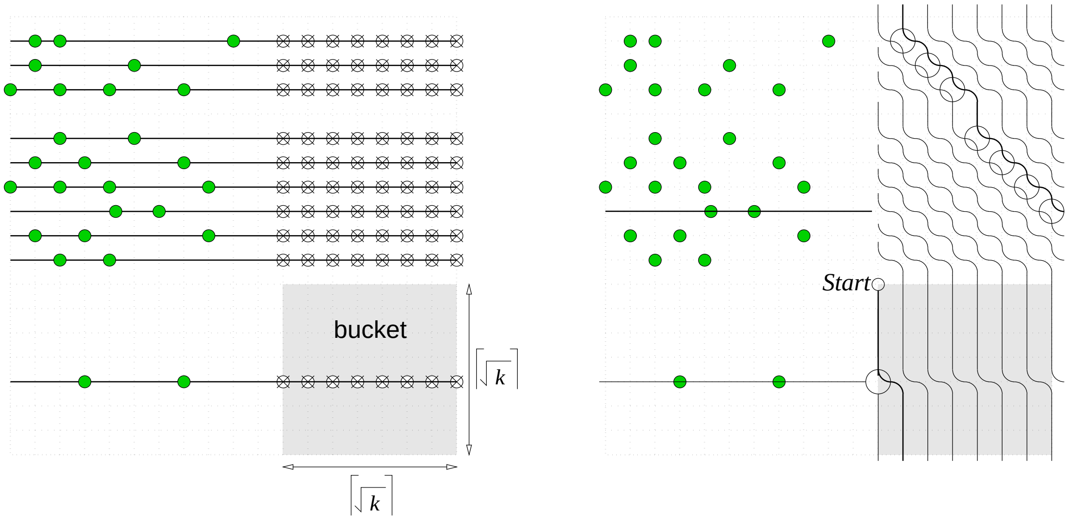

Figure 7: Steps 2.1.1 and 2.1.2 of the k-Packing Algorithm

In step 2.1.2, a subset of the rows identified in step 2.1.1 are matched with unique bucket columns. This is achieved using a simple algorithm, first introduced in [5], which operates as follows. All PEs who detected a signal in step 2.1.1 (those PEs who exist in a bucket column and a row which contains an element) configure (NE,SW) (as shown in Figure 7 pane 2) and all others configure (NS,EW). The row busses are cut at column $n-1$ so that no horizontal rings exist. The first PE in the read-set (denoted the *start* PE) sends a binary signal to the South. Any PE which detects this signal to the North sets a flag and represents the "matched" intersection between a row containing an element and a bucket column (these PEs are marked with a circle in Figure 7 pane 2).

In the next step, step 2.1.3, these intersection points are used to move an element from one of the sources in each matched row to the read PE in the corresponding (ie. 'matched')column. This operation is illustrated in Figure 8 pane 1. This is easily done - the intersection PE simply configures (NE,SW) and all others configure (NS,EW). All source PEs cut the row bus, send their element to the East, and attempt to read an element from the West. All members of the read-set attempt to read an element from the North.

Note that for matched rows, one PE will loose an element - the element received by the reader. All unmatched rows will shift the elements around the horizontal ring bus, but each source PE will receive an element - no net change in that row (as far as element numbers) will be experienced.

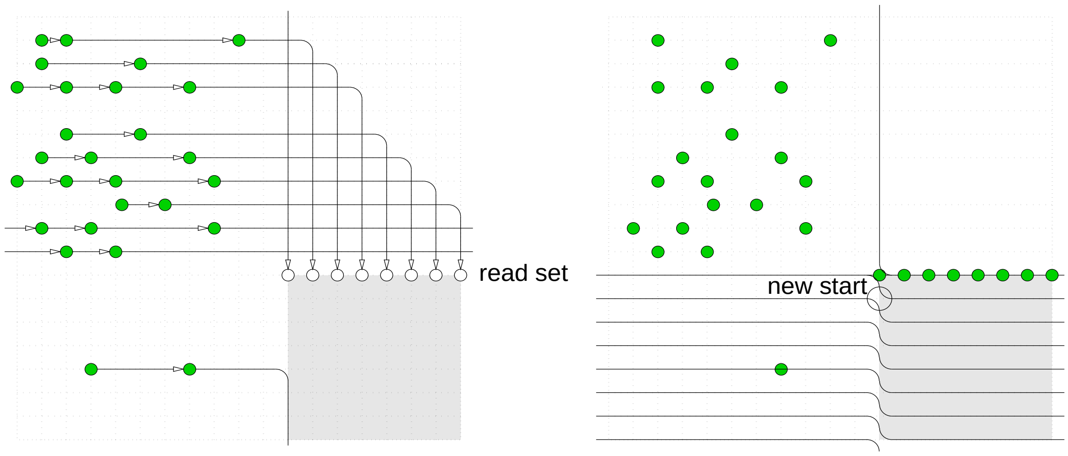

Figure 8: Steps (2.1.3 and 2.1.4), and (2.1.5 and 2.1.6) of the k-Packing Algorithm

5.2 Updating the Read Set

Once some number of readers have received an element, the read-set must be updated ready for the next phase. This is a simple process, requiring a small number of constant time operations. Firstly the new start PE, being the first bucket PE (based on a row-major ordering) which has not yet received an element, must be identified. This is achieved in one communication step and is illustrated in Figure 8 pane 2. All PEs in the column of the old start PE configure (NE,SW), all other bucket PEs configure (N,S,E,W) and all others configure (NS,EW). All PEs which received an element in the previous phase send a 1 to the East. All other bucket PEs send a 0, and attempt to detect a signal from the West (step 2.1.4). The PE which wrote a 0 to the East, but detected a 1 to the East is the new start PE (step 2.1.5).

Secondly, the new read-set must be determined. This is quite simple and may be achieved in 1 communication step. All PEs configure (N,S,E,W), and those PEs which received an element in the previous phase send a 1 to the South (step 2.1.8). This is illustrated in Figure 9. Those PEs which receive the 1 from the North add themselves to the read-set (step 2.1.9). Those PEs who were already members of the read-set and did not receive an element in the previous phase remain in the read-set. Those who received an element remove themselves from the read-set.

Lastly, vertical rings are created in the bucket columns, and the start-column is identified by the start PE broadcasting a signal along the column (step 2.1.7).

5.3 The Vertical Phase

The vertical phase is identical to the horizontal phase (steps 2.1.1 to 2.1.9) except that columns containing elements are matched with bucket columns. This can be easily achieved by using the horizontal approach, but prepending it with a step which collects the elements from the first $\lceil\sqrt{k}\rceil$ columns along the mesh diagonal, and then operates using the corresponding rows to transfer the elements to the read-set. This

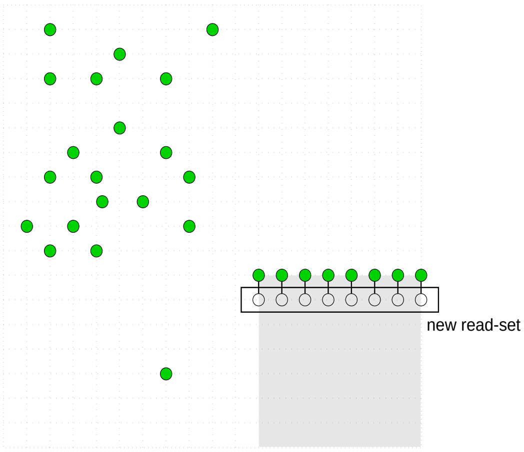

Figure 9: Step 2.1.8 and 2.1.9 of the k-Packing Algorithm

adds only one extra communication step to the process, ensuring that both the horizontal and vertical phases of the algorithm may be implemented in constant time.

5.4 Analysis

We have described how each of the horizontal and vertical phases are completed in constant time. If we observe that for each horizontal-vertical phase pair we must have at least $\lceil\sqrt{k}\rceil$ rows which contain elements OR $\lceil\sqrt{k}\rceil$ columns which contain elements, then after $\lceil\sqrt{k}\rceil$ iterations of the process (or $2\times\lceil\sqrt{k}\rceil$ phases in total) we will have moved all elements into the bucket. Therefore the algorithm is in $\mathcal{O}(\sqrt{k})$, meets the bisection lower bound, and is worst-case optimal.

5.5 Reversing the Algorithm

To be able to reverse the process and remove all elements from the bucket back to their original locations, we must modify the process slightly. In addition to the source address of the element, we also store the phase number at which the element was moved into the bucket. With the phase number stored, to reverse the element packing we must simply retrace our steps. Packets which were moved into the bucket in the last (vertical) phase are moved out of the bucket along their bucket column (note that no two elements with the same phase number can be in the same bucket column) to the correct row, and then the correct column. Similarly elements from subsequent phases are routed back to their original location, and the routing for the elements of each phase is completed in 3 steps for the reversal of vertical phases, and 2 steps for the reversal of horizontal phases. Since there are at most $\mathcal{O}(\lceil\sqrt{k}\rceil)$ phases to reverse, this takes time in $\mathcal{O}(\lceil\sqrt{k}\rceil)$.

6 Conclusion

In this paper we have proposed an algorithm to solve the so-called *sparse token-distribution* problem on reconfigurable meshes. This is the first algorithm to be proposed for this problem, and optimally reduces the discrepancy Δ by 1 for both k-sparse and non-sparse distributions. For k-sparse distributions the algorithm operates in worst-case optimal $\mathcal{O}(\sqrt{k})$ time. For general distributions, ie. those for which $k \in \mathcal{O}(n^2)$, the algorithm operates in time $\mathcal{O}(n)$ - meeting the bisection bound.

Future work aims at extending the algorithm for arbitrary dimensions and modifying it to operate under more restrictive models of reconfiguration - specifically under the restriction of fixed bus length.

References

[1] Schröder, H. and Turner, G. 'Token Distribution on Reconfigurable d-Dimensional Meshes with Restricted Bus Length', To appear: Proceedings of the 12th Australasian Computer Science Conference 1997.

[2] Houle, M. E. and Turner, G., 'Dimension-Exchange Token Distribution on the Mesh and Torus', *Technical Report CS-TR-96/8*, Department of Computer Science, Victoria University of Wellington, New Zealand.

[3] Kunde, M. and Gürtzig, K., 'Efficient Sorting and Routing on Reconfigurable Meshes Using Restricted Bus Length', *In Submission.*

[4] Kaufmann, M., Schröder, H., and Sibyen, J., 'Fast Sorting and Routing on Reconfigurable Meshes', *Parallel Processing Letters*, Vol 5, 1995, pp81-95.

[5] Turner, G. and Schröder, H., 'Token Distribution on Reconfigurable d-Dimensional Meshes', *Parallel Algorithms and Applications*, Vol 8, 1996, pp 251-269.

[6] J.W. Jang, H. Park and V.K. Prasana, 'A bit model of the reconfigurable mesh', *In Proceedings of Reconfigurable Architecture Workshop.* IEEE, April 1994.

[7] Lie, K.T., Schröder, H., 'A Reconfigurable Mesh Processor Array Utilising Fault-Tolerant Network', *PRFTS 93*, Melbourne, December 1993.

[8] Peleg, D., Upfal, E., 'The Generalised Packet Routing Problem', *Theoretical Computer Science*, 53, pp218-293, 1987.