Synchronizing Arbitrary Processor Groups in Dynamically Partitioned 2-D Meshes *

George Chochia, Murray Cole

Department of Computer Science, University of Edinburgh.

e-mail: {gac,mic}@dcs.ed.ac.uk

Todd Heywood

IBM PowerParallel Systems

e-mail: theywood@kgn.ibm.com

Abstract

A general purpose synchronization mechanism for a parallel computer should allow arbitrary, data-dependent, dynamically partitioned groups of processors to remain internally synchronized, while proceeding asynchronously with respect to other groups. We present an algorithm which can support such a scheme. The algorithm constructs binary synchronization trees for the sub-groups, given a group of processors and a $\{0,1\}$ label for each processor, and is valid for any network. We provide a general complexity analysis in terms of operations on the synchronization trees which is then instantiated with respect to the $n \times n$ processor 2D mesh architecture. We show that the algorithm constructs a synchronization tree for any sub-group of s processors in $O(n \log s)$ parallel communication steps with high probability. We present lower bounds on achievable performance based on the mesh indexing scheme used: row/column major indexing schemes require $\Omega(n \log n)$ parallel communication steps in the worst case, whereas the recursive Hilbert indexing scheme requires $\Omega(n \sqrt{\log n})$ parallel communication steps. Experimental results are given validating the analysis. Our algorithm has applications in implementations of PRAMs (e.g. conditional instructions) and of nested data parallelism (or mixed data/task parallelism) on distributed processor networks.

*Work supported by EPSRC grant GR/J43295

1 Introduction

The PRAM [7] is a well-known, powerful, idealized model of parallel computation, in which tightly synchronized processors share access to a large common memory. The H-PRAM [6] is a variant of the standard PRAM which allows arbitrary, recursive partitioning of the PRAM (both processors and memory), with a cost-model which charges communication and synchronization costs in proportion to the size of each sub-machine, thereby encouraging the algorithm designer to contrive algorithms in which as much work as possible takes place at lower levels of the resulting hierarchy, thus reducing overhead while retaining the ease-of-use features of the PRAM. Correspondingly, the implementer of the model has the task of devising strategies at the physical level which respect the intended locality of activity represented by the partitioning into groups.

The work described here was generated by a larger investigation into the practical details of H-PRAM implementation techniques for two dimensional mesh architectures. Since the H-PRAM subsumes the PRAM model as a special case, many of the implementation issues are common to both models, with additional complications sometimes arising through the added arbitrary partitionability of the H-PRAM. The problem described and solved in this paper is one such instance.

The problem is that of implementing multiple tightly synchronized sub-groups of processors, where different sub-groups are asynchronous/independent, which are created dynamically by partitioning a group of processors based on run-time data values. Although this is necessary for H-PRAM partitioning, note that it is also a basic requirement for implementing conditionals in a PRAM programming language.

Additionally, stripped of its (H-) PRAM motivation, the sub-group synchronization problem is also of interest to a more general class of parallel programming models mapped to distributed network architectures; one example is that of nested data parallel languages[3], which are more amenable to producing efficient programs for irregular problems than flat data parallelism.

The paper is organized as follows. The remainder of this section gives the background for the problem. In Section 2 we give bounds on the properties of binary synchronization trees embedded in 2D meshes with respect to different indexing schemes. In particular, we use path length arguments to show that Hilbert mesh indexing admits the possibility of a $\Theta(n\sqrt{\log n})$ synchronization algorithm, an improvement on the the $\Omega(n \log n)$ lower bound for row-major and snake-like indexing schemes which can be deduced by similar arguments. In Section 3 we present and prove the correctness of the algorithm for "sub-group identification", where a group of mesh processors is dynamically partitioned into arbitrary sub-groups, and synchronization trees constructed for the sub-groups. Section 4 contains the complexity analysis of the algorithm. We prove that on a $n \times n$ processor mesh any sub-group can be identified (synchronized) in $O(n \log n)$ parallel communication steps with high probability. Section 5 presents the results of simulated executions of the algorithm on the 2D mesh. We observe that for contiguous sub-groups, Hilbert indexing gives rise to significantly better performance than row-major indexing. Section 6 discusses related work.

1.1 Background

Conventional descriptions of the PRAM model in algorithms texts [7] state that the processors, while free to execute arbitrary and independent sequences of instructions, are nevertheless synchronized after each such instruction (i.e. in 'lock-step' style). This is a satisfactory description for the purposes of informal algorithm design and analysis. However, as soon as one attempts to build a conventional higher level programming model it becomes clear that the semantics of synchronization must be tied to the language constructs themselves (and may, but are not required to, be tied to lower level hardware synchronization support). The Fork language [5] and its variants represent the best developed such framework of which we are aware (our own experiments have been based on extending the concepts provided by Fork to the H-PRAM).

In Fork, all processors initially belong to the same 'group' which means that they are executing the same sequence of statements, in a tightly synchronized manner (with synchronization points after each statement, between evaluation of right hand sides, left hand sides and actual assignment in assignment statements and so on). Execution of a conditional statement may complicate this scenario. If the conditional expression evaluates differently in different processors (for example by depending upon processor identifiers or private data), then two sub-groups are formed, one for those processors choosing the `then` branch, the other for those choosing the `else`. These groups then execute the statements from their chosen branch independently, and asynchronously (the processors *within* a sub-group remain synchronous in the usual way). The two sub-groups are eventually recombined and synchronized when both branches of the conditional have terminated. In the context of a simple PRAM to mesh implementation, in which each physical processor emulates a single PRAM processor, this results in the following problem.

> Devise an efficient scheme which allows arbitrary and dynamically emerging sub-groups of processors to *identify* and *synchronize* themselves. The only available information on group composition will be that each group is a sub-set of an existing group and that the processors in each group are identified by the common result of an immediately preceding boolean expression evaluation.

In the H-PRAM context the problem is further complicated by the fact that these 'PRAM conditional' concerns may be occuring within an already partitioned (and irregularly shaped) sub-area of the mesh. Therefore, our interest is in sub-group synchronization within arbitrary groups of mesh processors.

It is important to emphasize at this point that the problem of implementing the H-PRAM specific partition is orthogonal to the problem discussed here. The 'sub-group' hierarchies discussed in this paper are artefacts of the PRAM model itself, and emerge and disappear dynamically within each sub-PRAM (sub-group) of an H-PRAM. This raises a challenging issue in the area of H-PRAM language design – that of nesting partitions within conditionals – for the moment, we hold the view that H-PRAM partitioning steps should be prohibited while any 'conditional' sub-group structure is in operation, and our method is constructed under this assumption.

2 Tree synchronization of arbitrary sub-meshes

In this section, we consider the properties of synchronization trees embedded in 2D meshes, producing performance bounds for synchronization based on binary trees.

Let U denote a set of 2D mesh nodes. Each node is identified by a pair of Cartesian coordinates (u_x, u_y). We denote the *row major*, *column major*, *snake-like* and *Hilbert* indexing schemes as I_Y, where $Y = \{R, C, S, H\}$ respectively (where the subscript is omitted, results apply to any of the indexing schemes). The first three schemes are well-known. For row major, given any pair $u, v \in U$ such that $I_R(u) > I_R(v)$, then either $u_y > v_y$ or $(u_y = v_y) \wedge (u_x > v_x)$. By swapping x and y, we obtain the column major indexing scheme I_C. For the snake-like scheme, $I_S(u) > I_S(v)$ iff either $u_y > v_y$ or $(u_y = v_y) \wedge (u_x > v_x) \wedge (u_y \text{ is even})$ or $(u_y = v_y) \wedge (u_x < v_x) \wedge (u_y \text{ is odd})$.

The Hilbert indexing scheme is relatively new (e.g. see [8] and [4], where the name "Peano indexing" is used, inaccurately). I_H can be defined recursively for any $2^q \times 2^q$, $q > 0$ mesh by the process shown in Figure 1.

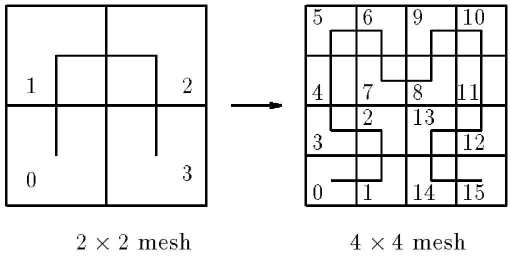

Figure 1: *The recursive rule for Hilbert indexing scheme. Each node of the mesh is shown as a square.*

We will use the following fact later. Here, the Manhattan distance (later distance) between two nodes u, v of a mesh is $d(u,v) = |u_x - v_x| + |u_y - v_y|$.

Fact 1 *[4, Theorem 1]. For any $u, v \in U$ within a mesh*

$$d(u,v) < 3\sqrt{|I_H(u) - I_H(v)|} \quad .$$

We now consider embedding synchronization trees in the mesh. Given a tree with a set of vertices T, $|U| = |T|$, introduce a bijective mapping on the mesh nodes and tree vertices: $U \leftrightarrow T$. For any $u \in U$ and $v \in T$ such that $v \leftrightarrow u$, put $I(u) = I(v)$. In the following we use "almost balanced" binary trees, where all intermediate vertices have two children except for some which are parents of leaf vertices. When tree edges are directed from the leaves to the root we have a combining tree, and when from root to leaves a broadcasting tree. Given a mapping $U \leftrightarrow T$, the synchronization of the mesh nodes can be performed in two passes, first on a combining tree and then on a broadcasting tree:

1. wait for packets from the children, then send a packet to the parent vertex;

2. receive a packet from the parent, then broadcast it to the children.

Any sub-group of U can be synchronized via a binary tree. The constant vertex degree translates to a constant space requirement for storing parent/children information.

The mapping $U' \leftrightarrow T$, where $U' \subseteq U$ and $|U'| = |T|$, is defined as follows. Let $rank\, I(v)$, $v \in U'$ be the rank of v with respect to indexing scheme I. Select

$u \in U'$, such that $\operatorname{rank} I(u) = \lceil (\max_v \operatorname{rank} I(v))/2 \rceil$, $v \in U'$. Split U' into two subsets $U'_<$ and $U'_>$ of nodes with ranks $>$ rank $I(u)$ and $<$ rank $I(u)$ respectively. Find left child in $U'_<$ and right child in $U'_>$ applying the described procedure recursively. Choose the node found at the first step as a root.

Assertion 1 *Given a sub-group $U' \subseteq U$, $s = |U'|$ and the rank $r = [0, s)$ of a vertex, the ranks of the vertex's children and parent in T, $U' \leftrightarrow T$, may be found in $O(\log s)$ computational steps.*

Proof For any two indexing schemes I and I' where $I(v) < I(u)$ iff $I'(v) < I'(u)$ for all distinct $v, u \in U'$, the mapping procedure finds the same children for a given parent. Define $I'(u) = \operatorname{rank} I(u)$, which introduces a total ordering on U'. Applying the mapping procedure with respect to I' we find a vertex with rank r in $O(\log s)$ computational steps. $O(1)$ additional steps find the ranks of its children and its parent. ∎

We will use the following properties of the mapping $U' \to T$.

Lemma 1 *For any $u, v \in T$, $s = |T|$ such that u is a parent of v, and there are $k \geq 0$ edges between u and the root,*

$$|rank\, I(u) - rank\, I(v)| \leq \left\lceil \frac{s}{2^{k+2}} \right\rceil \quad .$$

Proof Consider the case $k = 0$. Let u, v_L and v_R be the root and its left and right children, with ranks a, a_L, a_R respectively. Then $a = \lceil (s-1)/2 \rceil$, $a_L = \lceil (a-1)/2 \rceil$ and $a_R = a + \lceil (s-a)/2 \rceil$. Consider the differences $|a - a_L|$ and $|a - a_R|$. If a is even then $|a - a_L| = a/2$ and $|a - a_R| = \lceil (s-a)/2 \rceil$, which is $a/2$ if s is even and $a/2 + 1$ if s is odd. Indeed, if s is even $a = s/2$, $|a - a_R| = \lceil a/2 \rceil = a/2$; if s is odd $a = (s-1)/2$, $|a - a_R| = \lceil (a+1)/2 \rceil = a/2 + 1$. The right hand side is $\lceil s/4 \rceil = \lceil a/2 \rceil = a/2$ if s is even and is $\lceil s/4 \rceil = \lceil (2a+1)/4 \rceil = a/2 + 1$ if s is odd, therefore the inequality holds if a is even. If a is odd then $|a - a_L| = (a+1)/2$ and $|a - a_R| = \lceil (s-1)/2 - (a-1)/2 \rceil = \lceil (s-1)/2 \rceil - (a-1)/2 = (a+1)/2$. The right hand is $\lceil s/4 \rceil \geq \lceil a/2 \rceil = (a+1)/2$ and the inequality holds again. Hence the inequality holds for any s if $k = 0$. The size of $|U_<|$ and $|U_>|$ is at most $\lfloor s/2 \rfloor$ at step $k = 1$. At step $k \geq 1$ the size of these subsets s_k is at most $s_k \leq \lfloor s/2^k \rfloor$. Hence at step k the right hand is $\lceil s_k/4 \rceil$. Since the inequality holds for any s if $k = 0$, it holds for $s = s_k$. Applying $\lceil s_k/4 \rceil \leq \lceil s/2^{k+2} \rceil$ we conclude that Lemma holds at step $k \geq 1$. ∎

Assertion 2 *Vertices in the left (right) sub-tree rooted by u have smaller (larger) ranks than u. For any vertex u in the left (right) sub-tree there exists vertex v, such that $rank\, I(u) - rank\, I(v) = 1$, ($rank\, I(v) - rank\, I(u) = 1$).*

Proof From the definition of the mapping procedure it follows that, for any vertex u, its left and right children are in subsets $U'_<$ and $U'_>$ respectively. If non-empty, $U'_<$ contains node v with maximal index less than $I(u)$. Therefore $\text{rank}\, I(u) - \text{rank}\, I(v) = 1$. If non-empty, $U'_>$ contains node v with minimal index greater than $I(u)$. Therefore $\text{rank}\, I(v) - \text{rank}\, I(u) = 1$. ∎

The number of links a packet may cross on its way from a leaf node to the root on a synchronization tree varies, depending on the indexing scheme and the sub-group $U' \subseteq U$. The following theorem shows that the maximal number of links for the Hilbert indexing scheme is $\Theta(n\sqrt{\log n})$.

Theorem 1 *For a synchronization scheme based on a binary tree mapped to the Hilbert indexed mesh of size n^2 by the procedure described above, there exists a sub-group of nodes such that some synchronization packet crosses at least $\frac{n}{\sqrt{2}}\sqrt{\log n}\,(1+o(1))$ links. No packet crosses more than $3\sqrt{2}\,n\sqrt{\log n}$ links.*

Proof We start by proving the first statement. Split the mesh into sub-meshes of size $w = 4^q$, $0 \leq q \leq \log n$. Split each sub-mesh into four equal squares and mark the lower left one. Define S as the set of marked squares. Assign a rank to each square as they appear when moving along the Hilbert curve in increasing index order. For any $u, v \in S$, $u \neq v$ the distance between any pair of nodes, one from square u and the other from square v, is $\geq \sqrt{w}/2$.

We construct a sub-group in the following way. Mark one node from the square with rank zero, two nodes from the square with rank one, and so on doubling the number of nodes until either (a) the number of marked nodes in a square is equal to $w/4$ or (b) we run out of squares. Suppose q is small enough that (a) is the case. Then the sub-group contains $w/2 - 1$ elements. Define the set of marked nodes as the sub-group which is to be synchronized. The sub-group is mapped to a binary tree via the procedure described above. As a result we obtain a balanced binary tree of depth $\log w - 2$. From the recursive definition of I_H it can be seen that the nodes of $v \in S$ are consecutively numbered. Combining this observation with Assertion 2, we may conclude that in any square of rank k, $k < \log w - 2$, there exists a node having a parent in the square with rank $k+1$. Therefore, a synchronization packet generated by the marked node from the square with rank zero will need to cross at least $(\log w - 2) \cdot \sqrt{w}/2$ links. For our construction to be valid we need to have at least $\log w - 2$ elements in S. This is satisfied if $n^2/w \geq \log w - 2$, which (asymptotically) translates to

$$w = \frac{n^2}{\log n - 2}(1 + o(1)) \quad .$$

This value gives us the result that the packet generated in the square with rank zero crosses at least $\frac{n}{\sqrt{2}}\sqrt{\log n}\,(1 + o(1))$ links, thus proving the lower bound on the maximal number of links to be crossed.

We now proceed to the second statement in the theorem, the upper bound. Suppose a synchronization tree has at most k edges from some leaf to the root. Consider nodes $v_i \in U$, $i = 1, ..., k+1$, on the path from that leaf to the root. The number of links l crossed along the path is $\sum d(v_{i+1}, v_i)$. Defining $x_i = |I_H(v_{i+1}) - I_H(v_i)|$, $i = 1, ..., k$ and applying Fact 1 we find the upper bound on l via

$$\begin{aligned} l &\leq 3 \sum_{i=1}^{k} \sqrt{x_i} \\ \sum_{i=1}^{k} x_i &\leq n^2 \quad , \end{aligned}$$

from which we obtain

$$l = 3 \left(\sum_{i,j=1}^{k} \sqrt{x_i}\sqrt{x_j} \right)^{1/2} \quad \leq \quad 3 \left(\sum_{i,j=1}^{k} (x_i + x_j)/2 \right)^{1/2} \quad \leq \quad 3\sqrt{k}\,n \quad .$$

At the second step here, we have applied $\sqrt{x_i\,x_j} \leq (x_i + x_j)/2$. As we have at most n^2 nodes, $k \leq 2\log n$. This gives us the upper bound on the maximal number of links, $3\sqrt{2}\,n\sqrt{\log n}$. ∎

We now show that if the nodes within a sub-group are at the same stride r from each other, i.e. $|I_H(v_{i+1}) - I_H(v_i)| = r$, $i = 1, ..., |U'|$ for all $v_{i+1}, v_i \in U'$, such that $\mathrm{rank}\, I(v_{i+1}) = \mathrm{rank}\, I(v_i) + 1$, then any packet crosses at most $O(n)$ links. We call this set of nodes a *sub-group of equidistant nodes.*

Theorem 2 *Given a sub-group of s equidistant nodes on a Hilbert indexed mesh; if the stride $r = 1$ then no packet crosses more than*

$$3\,\alpha\,\sqrt{s} + O(\log s)$$

links; if the stride $r > 1$ then no packet crosses more than

$$3\,n\left(\alpha + O((\log s)/\sqrt{s-1})\right)$$

links, where $\alpha = (2+\sqrt{2})/2$.

Proof Consider case $r = 1$ first. The number of edges in T from any leaf to the root is at most $\lfloor \log s \rfloor$. Let v_i, $i = 0, ..., \lfloor \log s \rfloor - 1$ be a sequence of nodes maximizing the number of links $l = \sum d(v_{i+1}, v_i)$. As the nodes are consecutively numbered, we have $|I_H(v_{i+1}) - I_H(v_i)| = |\text{rank}\, I_H(v_{i+1}) - \text{rank}\, I_H(v_i)|$. By using Fact 1, and Lemma 1 we can bound l as

$$l \;\leq\; 3 \sum_{i=0}^{\lfloor \log s \rfloor - 1} \sqrt{\left\lceil \frac{s}{2^{i+2}} \right\rceil} \;\leq\; 3\left(\sum_{i=0}^{\lfloor \log s \rfloor - 1} \sqrt{\frac{s}{2^{i+2}}} + \lfloor \log s \rfloor \right) \;\leq\; 3\,\alpha\,\sqrt{s} + O(\log s)\ .$$

In this case the number of links crossed by any packet is at most $O(\sqrt{s})$.

Now consider Case $r > 1$. Here we have $|I_H(v_{i+1}) - I_H(v_i)| = r\,|\text{rank}\, I_H(v_{i+1}) - \text{rank}\, I_H(v_i)|$ and $r\,(s-1) \leq n^2$, from which we obtain $\sqrt{r} \leq \frac{n}{\sqrt{s-1}}$. Applying Fact 1 and Lemma 1 gives

$$l \;\leq\; 3 \sum_{i=0}^{\lfloor \log s \rfloor - 1} \sqrt{r \left\lceil \frac{s}{2^{i+2}} \right\rceil} \;\leq\; 3\,n\left(\alpha + O((\log s)/\sqrt{s-1})\right)\ .$$

■

Finally, we show that the structure of the other indexing schemes result in a worse lower bound on acievable performance (when compared to the result from theorem 1).

Theorem 3 *For a synchronization scheme based on a binary tree mapped to the row major or snake indexed mesh of size n^2 by the procedure described above, then there exists a sub-group of nodes such that some synchronization packet crosses $\Omega(n\,\log n)$ links.*

Proof First, consider the row-major indexing scheme (and by simple symmetry, the column major scheme). Let the number of nodes in a sub-group be $2^k - 1$, $n/2 \leq 2^k - 1 \leq n$. The mapping procedure described above constructs *a balanced* binary synchronization tree. Mark all parents at an odd distance from the leaves as o and the rest as e; see Figure 2. Now, in increasing order of their ranks, map all e marked vertices to mesh nodes on the left side of a mesh starting from the

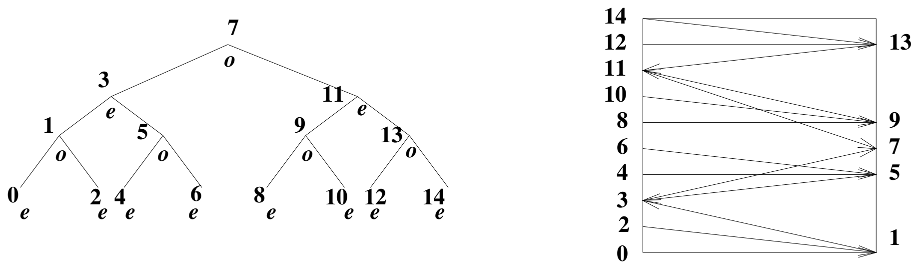

Figure 2: *Mapping of the vertices on the row-major indexed mesh nodes. A mesh node with Cartesian coordinates* $(0, 0)$ *is in the left bottom corner.*

node with Cartesian coordinates $(0, 0)$. Each new vertex is mapped on a leftmost (smallest indexed) mesh node on a new line. In increasing order of their ranks, map all o marked vertices to mesh nodes on the right side of a mesh, so that a vertex with rank r is on the same line with e marked vertex of rank $r - 1$ (from Assertion 2 we know that such a vertex exists). As we have no more than n vertices in a tree the construction is valid. Each node is at Manhattan distance at least $n - 1$ from its parent. The depth of a tree is $\Omega(\log n)$, therefore the number of mesh links crossed by packets issued at leaf nodes on their way to the root is $\Omega(n \log n)$.

Second, we address the snake-like indexing scheme. Let the number of nodes in a sub-group be $2^k - 1$, $n/4 \leq 2^k - 1 \leq n/2$. Map o indexed vertices as for row-major indexing and map e indexed vertices to the leftmost (smallest indexed) mesh nodes of *even* lines. Since we have no more than $n/2$ elements in a sub-group the construction is valid, which implies that $\Omega(n \log n)$ links will be crossed by packets issued at leaf nodes on their way to the root. ∎

3 The Sub-group Identification Algorithm

We now consider the problem of constructing synchronization trees for sub-groups from a dynamically split parent group. The sub-group identification problem can be formulated as follows. *Given*

1. a group of nodes U,

2. the mapping $U \leftrightarrow T$ to the tree vertices, and

3. the function $\varphi(u) \in \{0, 1\}$, $u \in T$, called a *vertex state*,

the *task* is to

1. split U into two non intersecting subsets U^0 and U^1: $U^x = \{u \in U | \varphi(u) = x\}$, $x = \{0, 1\}$,

2. map each subset on its new synchronization tree $U^x \leftrightarrow T^x$, applying the procedure described in the previous section, and

3. for each vertex $u \in T$ depending on its state, find the coordinates of its children and a parent in the new synchronization tree.

Once a sub-group is identified, the resulting synchronization tree is used to implement synchronous program steps. At some point, the sub-group program will generate either a new "sub-sub-group" identification requirement, or a requirement to restore the previous parent group. Subsequently, we call these basic operations *splitting* and *joining*.

We start with definitions which allow a subsequent concise, top-down description of the algorithm. Vertex $v \in T$ is called a *parent* of $v' \in T$ if it is an immediate predecessor of v'. If v is a predecessor of v' it is called an *ancestor* (so parents are also ancestors).

If node $u \in U$ is mapped on vertex $v \in T$, then $\varphi(v) = \varphi(u)$ and $I(v) = I(u)$. Let

$$\begin{aligned} P_L^x(v,u) &\stackrel{def}{=} (v \text{ is in the left sub-tree rooted by } u\,) \wedge (\varphi(v) = x) \\ P_R^x(v,u) &\stackrel{def}{=} (v \text{ is in the right sub-tree rooted by } u\,) \wedge (\varphi(v) = x) \quad . \end{aligned}$$

We use $z = \{L, R\}$ as a subscript for brevity. For any vertex $u \in T$, let subsets $F_z^x(u) = \{v \in T | P_z^x(v,u)\}$ which are the vertices in the left/right sub-tree rooted by u, satisfying $\varphi(v) = x$. Let $B^x(u)$ be the subsets of nodes with ranks smaller than u,

$$B^x(u) = \{v \in T | (\varphi(v) = x) \wedge (\,\mathrm{rank}\, I(v) < \mathrm{rank}\, I(u)\,)\}$$

Define functions: $t^x = |T^x|$, $f_z^x(u) = |F_z^x(u)|$, $b^x(u) = |B^x(u)|$, and

$$\Delta^x(u) = \begin{cases} u & \text{if } \varphi(u) = x; \\ \emptyset & \text{otherwise;} \end{cases}$$

We also use the function δ_j^i, $i, j \in \{0, 1\}$, which is unity if $i = j$ and zero otherwise. The new ranks of vertex $u \in T$ with respect to the indexing scheme I and state x are $\mathrm{rank}^x\, I(u) = b^x(u) + \delta_{\varphi(u)}^x - 1$. Note that the new ranks are defined for both states. As a consequence we have: if $\mathrm{rank}\, I(u) < \mathrm{rank}\, I(v)$ then $\mathrm{rank}^x\, I(u) \leq \mathrm{rank}^x\, I(v)$.

The function $m_z^x(u)$, $u \in T$ is defined as

$$m_z^x(u) = \max_{\{v \in T | P_z^x(v,u)\}} (\mathrm{rank}^x\, I(v), -\infty) \ , \tag{1}$$

which is the maximal rank of a vertex in state x in the left/right sub-tree rooted by u (and is defined to return $-\infty$ in case the maximum is taken over the empty set.

The sub-group identification algorithm has two phases. In the first phase we evaluate functions t^x, $f_z^x(u)$ and $b^x(u)$ in each vertex $v \in T$. In the second phase we find the coordinates of the children and the parent for all $u \in T$ on a new tree $T^{\varphi(u)}$. The first phase in itself consists of two passes. The forward pass is performed on a combining tree, the backward one on a broadcasting tree. The sub-groups belonging to the same parent group are called *adjacent*.

Our splitting and joining algorithms can be summarized as follows:

Sub-group identification (splitting)

1. *evaluate functions f_z^x on the forward pass, and functions t^x, b^x and m_z^x on the backward pass, as described in Lemma 2 below;*
2. *perform partial synchronization (forward pass);*
3. *identify ranks of the new parent and children (Assertion 1);*
4. *identify physical coordinates of the new parent and children;*
   if root, coordinates of a root in adjacent sub-group,
   applying algorithm in Figure 3;
5. *synchronize all vertices in the original tree;*
6. *activate new sub-group in each vertex.*

Restoring a sub-group (joining)

1. *perform partial synchronization (forward pass);*
2. *synchronize root vertices in adjacent sub-groups;*
3. *broadcast from the root to the leaves;*
4. *restore the original sub-group in each vertex.*

It must be understood that forward/backward pass and synchronization operations presume that a processor sends/receives some number (depending on its position in a tree) of packets to/from its parent and children. These "atomic" operations are performed asynchronously, which implies that processors in different nodes may be at different steps at any point in time.

We now present and justify the detailed implementations of the steps in the splitting algorithm. The joining algorithm is straightforward.

3.1 Calculating Ranks for the New Trees

In this subsection we show how the various rank-related functions can be computed for the new trees. This corresponds to step 1 in the splitting algorithm above.

Lemma 2 *The function f_z^x can be evaluated in the forward pass; the functions t^x, b^x and m_z^x can be evaluated in the backward pass.*

Proof It is sufficient to show that these function evaluations only require information stored in the vertex itself and its parent. The parent and children vertices are logically interchanged in the combining and broadcasting trees. Let us fix the orientation as it is for the broadcasting tree, i.e. from the root to the leaves. Then in the first pass it is sufficient to show that f_z^x requires the information stored in the vertex and its children; in the second pass t^x, m_z^x and b^x require the information in the vertex and its parent.

If $u \in T$ is a leaf, then $F_z^x(u) = \emptyset$, therefore $f_z^x(u) = 0$ is known in the leaves. Let u be an intermediate vertex or the root, and u_L, u_R be its left and right child respectively. Then $F_z^x(u) = F_L^x(u_z) \cup F_R^x(u_z) \cup \{\Delta^x(u_z)\}$, which means $f_z^x(u) = f_L^x(u_z) + f_R^x(u_z) + \delta^x_{\varphi(u_z)}$, as $F_L^x(u_z)$, $F_R^x(u_z)$ and u_z are pairwise non-intersecting sets. If v is the only right/left child of u, then either $f_L^x(u_z)$ or $f_R^x(u_z)$ corresponding to the left/right branch has to be dropped. Thus the information stored in the children vertices is sufficient for evaluation of $f_z^x(u)$ in the parent vertex u, therefore f_z^x can be evaluated in the forward pass.

If u is the root, $T^x = F_L^x(u) \cup F_R^x(u) \cup \{\Delta^x(u)\}$, and $t^x = f_L^x(u) + f_R^x(u) + \delta^x_{\varphi(u)}$ can be evaluated in the last step of the forward pass. Then t^x can be broadcasted to all vertices in the backward pass. It remains to show that b^x and m_z^x can be evaluated in the backward pass. If u is the root, $B^x(u) = F_L^x(u)$, therefore $b^x(u) = f_L^x(u)$ is known in the root by the end of the forward pass. Let v be the parent of u. Define a function $\eta(v,u) \in \{L, R\}$, such that $\eta(v,u) = L$ if u is the left child of v, and $\eta(v,u) = R$ otherwise; and also a subset $S_z^x \subset T$

$$\begin{aligned} S_R^x(u) &= S^x_{\eta(v,u)}(v) \cup F_L^x(u) \cup \{\Delta^x(u)\} \\ S_L^x(u) &= S^x_{\eta(v,u)}(v) \quad , \end{aligned}$$

where initially $S_L^x(u) = \emptyset$ and $S_R^x(u) = F_L^x(u) \cup \{\Delta^x(u)\}$ if u is a root. Call the first recursion the *right tree* rule and the second one the *left tree* rule. We claim that $B^x(u) = S_L^x(u) \cup F_L^x(u)$. Suppose the contrary. Then either (1) there exists $w \in T$ in state x with $\operatorname{rank} I(w) < \operatorname{rank} I(u)$, such that $w \notin B^x(u)$; or (2) $w' \in T$ in state x with $\operatorname{rank} I(w') > \operatorname{rank} I(u)$, and $w' \in B^x(u)$; or (3) $w'' \in T$ in state $\neg x$, and $w'' \in B^x(u)$. From the definition of F_L^x and Δ^x one may see that the third case is impossible.

Consider the first case. Let v' be the first common ancestor of u and w. From $\operatorname{rank} I(w) < \operatorname{rank} I(u)$ and Assertion 2 it follows that either $u = v'$ and w is in the left subtree rooted by u, or w is in the left subtree and u is in the right subtree rooted by v'. Consider the former. Then from $w \in F_L^x(u)$ and $F_L^x(u) \subseteq B^x(u)$ it follows $w \in B^x(u)$. Consider the latter. Let v'' be the right child of v'. Clearly, v'' is either a parent of u or $u = v''$. Thus we have $\eta(v', v'') = R$, $w \in F_L^x(v')$, $F_L^x(v') \subseteq S_R^x(v')$, $S_L^x(v'') = S_R^x(v')$, therefore $w \in S_L^x(v'')$. From $S_L^x(v) \subseteq S_R^x(v)$ for all $v \in T$, we have $w \in S_R^x(v'')$, hence $w \in B^x(u)$ regardless of which rule is applied in v''.

Consider the second case. Let v' be the first common ancestor of w' and u. From $\operatorname{rank} I(w') > \operatorname{rank} I(u)$ and Assertion 2 we conclude that either $v' = w'$, in which case u is in the left sub-tree rooted by w' or u is in the left subtree and w' is in the right subtree rooted by v'. In both cases $w' \notin F_L^x(u)$, therefore suppose $w' \in S_L^x(u)$. For this to be possible there must exist an ancestor v'' of u such that: (i) either $w' = v''$ or $w' \in F_L^x(v'')$, and (ii) the right rule is applied in v''. Therefore the recursive process follows the right branch in v''. This contradicts the fact that it reaches u which is in the left sub-tree of v''. Thus second case is also impossible.

It is not difficult to verify that if v is a parent of u then $S_{\eta(v,u)}^x(v)$, $F_L^x(u)$ and $\Delta^x(u)$ are pairwise non intersecting subsets for all $v, u \in T$. Indeed, from the definitions we have $\Delta^x(u) \cap F_L^x(u) = \emptyset$. Suppose $S_{\eta(v,u)}^x(v) \cap F_L^x(u) \neq \emptyset$. Then there exists v' such that $v' \in F_L^x(u)$ and $v' \in S_{\eta(v,u)}^x(v)$. The latter implies existence of v'', an ancestor of v, such that (i) $v' \in F_L^x(v'')$ and (ii) the right rule is applied in v''. Thus v' is in the left sub-tree rooted by u and it also is in the left sub-tree rooted by v'' which is an ancestor of v. This implies that v is in the left sub-tree rooted by v''. In order that the recursive process arrived in v it must have followed the left branch in v'', which contradicts (ii). Repeating similar arguments we may prove that $\Delta^x(u) \cap S_L^x(v) = \emptyset$. We are about to show that b^x can be evaluated on the backward pass. Let $s_z^x(u) = |S_z^x(u)|$, then as a consequence of set independence

$$s_R^x(u) \quad = \quad s_{\eta(v,u)}^x(v) + f_L^x(u) + \delta_{\varphi(u)}^x$$

$$s^x_L(u) \quad = \quad s^x_{\eta(v,u)}(v) \quad ,$$

with initial conditions $s^x_L(u) = \emptyset$ and $s^x_R(r) = f^x_L(r) + \delta^x_{\varphi(r)}$ in the root r. Therefore $b^x(u) = s^x_L(u) + f^x_L(u),\ u \in T$. Evidently, to evaluate $b^x(u)$ and $s^x_{\{L,R\}}(u)$ we only need data stored in the vertex i.e. $f^x_L(u)$ and $\delta^x_{\varphi(u)}$ and data passed from the parent vertex v i.e. $s^x_{\eta(v,u)}(v)$.

The function m^x_L can be evaluated as

$$m^x_L(u) = \begin{cases} b^x(u) - 1 & \text{if } f^x_L(u) > 0; \\ -\infty & \text{otherwise;} \end{cases}$$

We address the second condition first. From $f^x_L(u) = 0$ it follows that $F^x_L(u) = \emptyset$. From the definition of F^x_L, it follows that $P^x_z(v,u)$ is false for all $v \in T$, therefore the maximum in Equation (1) above is taken over the empty set, and $m^x_L(u) = -\infty$ as required. Consider the case $f^x_L(u) > 0$. $P^x_L(v,u)$ must be true for some $v \in T$. Let v be a vertex in state x in the left sub-tree of u such that $\operatorname{rank} I(v)$ is maximal. Recall the definition of $b^x(u)$. It counts the number of vertexes in state x having ranks (on the original tree) less than $\operatorname{rank} I(u)$. Choose vertex v' such that $\operatorname{rank}^x I(v') = b^x(u) - 1$. We claim that $v' = v$. Suppose the contrary. If v' is in the left sub-tree of u then $\operatorname{rank} I(v') < \operatorname{rank} I(v)$ and $b^x(v) = b^x(u) - 1$. On the other hand $\operatorname{rank}^x I(v) = b^x(v) + \delta^x_{\varphi(u)} - 1 = b^x(v)$, because $\varphi(u) = x$. That means $\operatorname{rank}^x I(v) = b^x(u) - 1$, i.e. a contradiction as v and v' are in the same state therefore $\operatorname{rank}^x I(v) \neq \operatorname{rank}^x I(v')$. If v' is not in the left sub-tree of u then either v is in the right sub-tree of v', or v' is in the left sub-tree and v is in the right sub-tree of their common ancestor. In both cases (Assertion 2) $\operatorname{rank} I(v') < \operatorname{rank} I(v)$ which leads to a contradiction as above. Hence $v' = v$ and $\operatorname{rank}^x I(v) = b^x(u) - 1$. As $\operatorname{rank} I(v)$ is maximal in the left sub-tree and $\operatorname{rank}^x I(v') < \operatorname{rank}^x I(v'')$ whenever $\operatorname{rank} I(v') < \operatorname{rank} I(v'')$ for any pair $v', v'' \in T$ of vertexes in state x we conclude that

$$m^x_L(u) = \max_{\{v \in T | P^x_L(v,u)\}} \operatorname{rank}^x I(v) = b^x(u) - 1 \quad ,$$

whenever $f^x_L(u) > 0$.

It remains to evaluate m^x_R:

$$m^x_R(u) = \begin{cases} b^x(u) + f^x_R(u) + \delta^x_{\varphi(u)} - 1 & \text{if } f^x_R(u) > 0; \\ -\infty & \text{otherwise;} \end{cases}$$

The additive terms $f^x_R(u) + \delta^x_{\varphi(u)}$ in the first condition account for a contribution of vertices in the right subtree rooted by u. The proof reduces to the arguments used for m^x_L. ∎

3.2 Locating New Parents and Children

We now present a description of the second phase of the splitting algorithm, in which culminates in the identification of the physical locations of the new parents and children for each vertex in the new trees. This corresponds to steps 2 to 4 of the splitting algorithm outlined earlier. To begin this phase, we must be sure that all the functions of Lemma 2 have been evaluated in all vertices. To ensure that evaluations have finished, we perform *a partial synchronization* (step 2), which is a synchronization pass from the leaves to the root. One pass is sufficient because functions t^x, b^x and m_z^x have been evaluated in a preceding pass from the root to the leaves.

We then evaluate the ranks of the children and the parent vertices in the new synchronization tree. From Assertion 1, we conclude that $t^{\varphi(u)}$ and $\text{rank}^{\varphi(u)}\, I(u)$ are sufficient to perform this step.

The next step is to identify the Cartesian coordinates of the children and the parent vertices in $T^{\varphi(u)}$ for all $u \in T$. Define send(c) as a communication operation on a mesh node which sends a packet to the node with coordinates c. Assume that a packet carries a triple (q, r, c): the state $q \in \{0, 1\}$ of a (source) vertex looking for its new parent and children; the rank r of a destination vertex in a new tree T^q; and Cartesian coordinates c of the mesh node associated with the source vertex. In Figure 3 we give the algorithm ("routing on a tree") for determining the mesh coordinates of the new parent and children. Parent, left.child, and right.child apply to vertex $v \in T$, and give the Cartesian coordinates of its parent and its children. The routing of a packet is done when the packet arrives at the vertex with rank r in state q (this check is omitted).

```
if b^q > r then
  if m_L^q = -∞ then send ( parent(v) );
  else if ( m_L^q ≥ r ) ∧ ( m_L^q - f_L^q < r ) then send ( left.child(v) );
       else send ( parent(v) ); fi;fi;
else if m_R^q = -∞ then send ( parent(v) );
     else if ( m_R^q ≥ r ) ∧ ( m_R^q - f_R^q < r ) then send ( right.child(v) );
          else send ( parent(v) ); fi;fi;fi;
```

Figure 3: *The Algorithm for routing on a tree (step four).*

Lemma 3 *The packet (q, r, c) arrives in the vertex u, such that $rank^q(u) = r$.*

Proof Suppose a packet is in some $u \in T$. If $b^q(u) > r$, the destination vertex is not in the right subtree rooted by u. If $m_L^q(u) = -\infty$, the left sub-tree contains no vertices in state q, hence the only direction is the parent vertex. If $(m_L^q(u) \geq r) \wedge (m_L^q(u) - f_L^b(u) < r)$ is true, the maximal rank of the vertices (in state q) in the left sub-tree is $\geq r$ and the minimal rank is $< r$. Recalling Assertion 2, and the fact that given any pair of vertices in the same state, their ranks on a new tree are linked by the same relationship as on the original tree, we conclude that in the left sub-tree there exists a vertex with rank r. If either of these conditions is false, a packet will be routed to the parent vertex. Indeed, if $m_L^q(u) < r$, the maximal rank of the vertices in the left sub-tree is $< r$, hence a packet must be routed to the parent. From $m_L^q(u) > r + f_L^q(u)$ it follows that all vertices in the left sub-tree have ranks $> r$. Hence, in that case the packet also must be routed to the parent.

The branch of the algorithm corresponding to the case $b^q \leq r$ is proven via similar arguments. ■

Step 5 ensures that the routing on a tree has been completed. On completion the new sub-group is activated (step 6) i.e. T becomes $T^{\varphi(v)}$ for all $v \in T$. Notice that a packet passes only once through any node, in other words, the algorithm routes a packet along the shortest path on a tree. An example of a subgroup identification problem illustrating the algorithm is given in the appendix.

4 Algorithm Analysis

We now analyse the sub-group identification algorithm of the previous section. Let us look at this algorithm in terms of packet routing. As building blocks it incorporates three problems: combining, broadcasting and routing on a tree. Each of these problem can be formulated as an algorithm on logical (virtual) tree. In order to find their complexity we are using the following scheme: simulate each of these algorithms on a physical tree isomorphic to the logical tree, then simulate the physical tree on a mesh.

The complexity of the sub-group identification algorithm is analysed when independent sub-groups perform synchronization/identification operations simultaneously in other words in a *dynamic context.* We prove that given a sub-group of s elements the communication complexity of our algorithm is $O(n \log s)$ in this dynamic context. Lemmas 4 and 5 below cover the complexity for routing on a tree (step 4 of the splitting phase).

Lemma 4 *For any $u \in T$ which is a root of a subtree with s vertices, the number of packets in step 4 crossing the edges incident on u for any fixed orientation is at most* $3 \log s$.

Proof Consider any $U^x \subseteq U$ and the mapping $U^x \leftrightarrow T^x$ introduced in Section 2. The algorithm in Figure 3 routes the packets along the shortest path on a tree. From this and Assertion 2 we conclude that a packet will be routed through some vertex $u \in T$ if either the source (destination) vertex belongs to the tree rooted by u and the destination (source) vertex does not, or the source (destination) is in one of u's sub-trees and the destination (source) is either in the other sub-tree or is u itself. Let $T_u \subset T$ be a subtree with a root u. Define $T_u^x = \{v \in T_u | \varphi(v) = x\}$, $r_{max}^x = \max_{v \in T_u^x} \mathrm{rank}^x\, I(v)$ and $r_{min}^x = \min_{v \in T_u^x} \mathrm{rank}^x\, I(v)$. Let us call (v, v'), $v, v' \in T^x$, where either v or v' is a parent of the other a *connected pair*. Define sets

$$
\begin{aligned}
V_R &= \{v \in T_u^x |\ \mathrm{rank}^x I(v') > r_{max}^x,\ v' \in T^x\} \quad , \\
V_L &= \{v \in T_u^x |\ \mathrm{rank}^x\, I(v') < r_{min}^x,\ v' \in T^x\} \quad , \\
V_T &= \{v, v' \in T_u^x |\ (\mathrm{rank}^x I(v) \leq \mathrm{rank}^x I(u)) \wedge (\mathrm{rank}^x I(v') > \mathrm{rank}^x I(u))\} \quad .
\end{aligned}
$$

Notice that $|V_R| + |V_L| + |V_T|$ is the number of packets crossing edges incident on u for any fixed orientations. Find $w, w' \in V_R$, such that w has minimal rank in V_R, and w' has minimal rank in $V_R \setminus \{w\}$. Introduce the notation $l_{u,v} = \mathrm{rank}^x I(u) - \mathrm{rank}^x I(v)$. We claim that

$$l_{w',w} \ \geq\ (\, r_{max}^x - \mathrm{rank}^x I(w)\,)\,/\,2 \quad . \tag{2}$$

Indeed, letting (w', v') be a connected pair, then

$$l_{w',w} \ \geq\ l_{v'w'} \ >\ r_{max}^x - \mathrm{rank}^x I(w') \tag{3}$$

The second (right) inequality takes place as $\mathrm{rank}^x I(v') > r_{max}^x$. The first (left) one, as it is shown below, follows from the definition of the mapping procedure. Consider vertex v such that (w, v) is a connected pair. Then we have

$$\mathrm{rank}^x I(w) \ <\ \mathrm{rank}^x I(w') \ <\ \mathrm{rank}^x I(v') \ <\ \mathrm{rank}^x I(v) \quad . \tag{4}$$

The first two inequalities follow from the previous definition. Let us prove the last one. Indeed, as (w, v) and (w', v') are connected pairs the only possibilities are that (i) either none of vertexes from one pair is a parent of vertexes in the other (ii) or at least one vertex of four is an ancestor of the others. Case (i) presumes that (w, v) and (w', v') have a common ancestor. From Assertion 2 it follows that segments $[\text{rank}^x I(w), \text{rank}^x I(v)]$ and $[\text{rank}^x I(w'), \text{rank}^x I(v')]$ are non-intersecting, that is w and w' can not be members of V_R simultaneously, i.e. a contradiction. Therefore assume case (ii) takes place. Observe that $\text{rank}^x I(v) < \text{rank}^x I(v')$ leads to a contradiction: choosing any of v, w, v' or w' as an ancestor it is impossible to construct a sub-tree which is in agreement with Assertion 2. The only possible cases are: (a) w is the left child of v and w', v' are in the right subtree of w, (b) v is the right child of w and w', v' are in the left subtree of v.

Suppose that w' is a parent of v' (which is its right child). Let v'' be its left child. In both cases (a) and (b) above v'' is in the right subtree of u which means $l_{w',w} > l_{w',v''}$. From the definition of the mapping procedure we can see that for any sub-tree the number of vertexes in the left sub-tree is at most that in the right sub-tree plus one. For the case under consideration this can be written as $l_{w',v''} \geq l_{v',w'}$. Comparing with the preceding inequality we find that $l_{w',w} > l_{v',w'}$ therefore the first inequality in (3) holds if w' is a parent of v'.

Suppose v' is a parent of w'. From the definition of the mapping procedure we conclude that $l_{w',w} \geq l_{v',w}$ therefore both inequalities in (3) are proved. Comparing $l_{w',w} + r^x_{max} - \text{rank}^x\, I(w') = r^x_{max} - \text{rank}^x\, I(w)$ with (3) we obtain (2).

The process can be repeated with respect to w' and $w'' \in V_R$, where w'' has a minimal rank in $V_R \setminus (\{w\} \cup \{w'\})$. Applying (2) to $l_{w,w'}$, $l_{w',w''}$, etc, repeatedly we find that $|V_R| \leq \log\, r^x_{max} - r^x_{min} \leq \log s$. Similarly we prove that $V_L \leq \log s$ and $V_T \leq \log s$. Hence $|V_R| + |V_L| + |V_T| \leq 3\, \log s$. ∎

Our next step is to establish a bound on the number of the communication steps required for routing on a tree (step 4 of the splitting algorithm) on a network in the form of a binary tree. To avoid confusion we call a synchronization tree a *virtual tree* and the network a *physical tree*. A virtual tree is isomorphic to a physical tree. Assume that the physical tree has bidirectional links capable of transferring a packet between two adjacent vertices in one communication step.

Lemma 5 *Step 4 defined on a virtual tree T of size $s = |T|$ can be simulated on a physical tree of size s in $O(\log s)$ tree communication steps with queue size $O(\log s)$ in each node.*

Proof Let us represent step 4 as six routing problems solved one by one. These routing problems involve finding the parent and the left and right children in a new synchronization tree for each vertex in state zero and one. Each packet carries the rank of the destination node. Introduce the following prioritization scheme: (a) if there are two packets in a queue such that both have the rank of a source smaller than that of their destination then a packet with higher destination rank has priority over that with smaller rank; (b) if both packets have ranks of a destination smaller than that of a source then a packet with smaller destination rank has priority over that with a large one; (c) if one packet has a rank of a source smaller than that of a destination and the other one has a rank of a source larger than that of a destination then the first packet has priority over the second one. We refer to these as rule (a), (b) and (c) respectively. Let v and v' be a source and a destination such that $\mathrm{rank}^x I(v) < \mathrm{rank}^x I(v')$. Then v is in the left sub-tree rooted by some node u and v' is in the right sub-tree (we allow $u \equiv v'$).

Consider delays on the path from v to u. According to rule (a) this packet can be delayed only by packets having larger ranks of a destination. From $\mathrm{rank}^x I(v) < \mathrm{rank}^x I(v')$ we have $\mathrm{rank}\, I(v) < \mathrm{rank}\, I(u) < \mathrm{rank}\, I(v')$, which means that all packets with ranks of a destination larger than $\mathrm{rank}^x I(v')$ in a sub-tree rooted by u pass through the (left) link incident u. Let s_u denote the total number of nodes in a sub-tree rooted by u. From Lemma 4 we know that the total number of packets crossing this link is less than $3 \lceil \log s_u \rceil$. Assume that the queue size is $3 \lceil \log s_u \rceil$ for each link. Therefore the packet can not be delayed more than $3 \lceil \log s_u \rceil$ times on its way to u. When in node u, it can be delayed at most $3 \lceil \log s_u \rceil$ times by the packets leaving u along the right link, i.e. it may suffer at most $6 \lceil \log s_u \rceil$ delays in total. The distance from u to v' is at most $\lceil \log s_u \rceil$ hence nothing may prevent a packet from leaving u later than $7 \lceil \log s_u \rceil$.

Applying similar arguments we can prove the same result for $\mathrm{rank}^x I(v) > \mathrm{rank}^x I(v')$ (in this case a packet will be in the right sub-tree with respect to u, i.e. rules (b) and (c) are applicable on the path from v to u). Thus this property holds for all packets.

Consider delays on the path from u to v'. We claim that if the packet arrives in vertex u' (in the right sub-tree of u) with $s_{u'}$ nodes at time $7 \lceil \log s_u \rceil + \mathit{level}\, u' - \mathit{level}\, u$ it will not suffer additional delays. Suppose contrary. Then there must exist a delaying packet having a source and a destination within a subtree rooted by u'. As was proved above nothing may prevent the delaying packet from leaving u' by the time $7 \lceil \log s_{u'} \rceil < 7 \lceil \log s_u \rceil + \mathit{level}\, u' - \mathit{level}\, u$, which is a contradiction. Hence the packet arrives in v' by $8 \lceil \log s_u \rceil$ communication steps.

The outlined scheme works for all six routing problems mentioned above. Each routing problem can be solved on a physical tree in $O(\log s)$ tree communication steps. Solving the problems one by one we finish in $O(\log s)$ tree communication steps. ∎

Consider the simulation of physical tree on a mesh. It should be clear from the construction in Section 3 that for any mesh indexing scheme all vertices of a synchronization tree are mapped to different mesh nodes. Each node of a tree has at most three links associated with it. These facts mean that parallel communication on physical tree can be simulated as three $1-1$ permutation routing problems on a mesh, the first permutation associated with left child communication, the second with right child and the third with the parent. In [12] it has been shown that $1-1$ permutation routing on a mesh can be solved in $2n + O(\log n)$ parallel communication steps with $O(\log n)$ queue sizes with high probability (this algorithm has an adaptation preserving the *individual locality*, i.e. if a packet has to travel the distance d it will be routed in $O(d)$ communication steps). From Lemma 5 it follows that we need $O(\log s)$ parallel communications on physical tree to simulate step 4 (routing on a tree). Therefore we need at most $O(n \log s)$ parallel communication steps on a mesh. The same is true for combining and broadcasting. Indeed, given a sub-group of s elements these problems require $O(\log s)$ steps on a virtual tree and $O(\log s)$ parallel communications on physical tree.

Consider sub-group identification in dynamic context. Synchronization trees corresponding to these problems are mapped on different mesh nodes. Therefore simulation of parallel communications performed on these trees reduces to at most three $1-1$ permutations on a mesh. The above arguments provide a proof for

Theorem 4 *The algorithm identifies $\{0,1\}$ sub-groups within s-element sub-group in $O(n \log s)$ parallel mesh communication steps with queue size $O(\log n)$ in the dynamic context with high probability.*

Note that we did not exploit the locality preserving property of the permutation routing algorithm [12], which has only been proven to hold for a single permutation rather than for $1-1$ permutations performed in the dynamic context. However, this is likely to be the case in the dynamic context as well [13]. If so, we can state that the algorithm identifies a sub-group of any size on a Hilbert indexed mesh in $O(n\sqrt{\log n})$ communication steps in the dynamic context with high probability.

5 Experimental Results

In this section, we back up the analytical results with three sets of experiments:

1. Sub-group identification or synchronization tree construction for dynamically created sub-groups: overhead v. size of sub-group (Hilbert indexing).

2. Synchronization of sub-groups using dynamically created trees: overhead v. sub-group size (Hilbert indexing).

3. Sub-group identification for sub-groups consisting of consecutively indexed mesh processors: overhead v. sub-group size of (Hilbert and row-major)

All experiments use a 256 processor mesh and randomly generated sub-groups. The overhead is given as a number of parallel neighbour-to-neighbor mesh communications. Since we are interested in getting an idea of how the techniques will work in practice, and confirming the analytical results, the simulated mesh model is simple and realistic. Each input/output link has q buffers to store packets. Queue overflow is avoided by means of a "network stabilization" mechanism, which can be briefly summarized as follows: if there is a free buffer on the receiving link an acknowledgment is sent back to the sending link in the adjacent node, and the packet is sent; if there is no space the acknowledgment does not return until space appears (deadlock free routing guarantees that this happens eventually).

The protocol behind the experiments, with respect to the dynamic generation of sub-groups (i.e. their 0/1 labels, for input to the splitting process) needs to be outlined. We specify which processors become members of which sub-groups as follows. At the top most level all nodes belong to one group. Each processor executes a loop where a fixed number of nested splittings is followed by the same number of joins. For each splitting a processor receives a random argument: either 1 or 0 identifying its membership in a sub-group (these values are the random variables). To collect statistics uniformly over the sub-group size we applied the following distribution. Consider a sub-group of size s at some level of splitting. Each processor evaluates two independent uniformly distributed random variables $\zeta, \xi \in [0, s)$. ζ is the same for all processors within a sub-group, while all ξ's are independent. If $\xi > \zeta$, then assign a processor to a sub-group 1, otherwise to 0.

The results for experiment sets 1 and 2 listed above are given in Figure 4. Experiments were repeated at least 2000 times for each case. The nesting (splitting) level is four. Note that the overhead grows within the region $s < n$ and remain almost constant afterwards. The worst case complexity of splitting overhead for Hilbert indexing given in section 2 can be generalized for sub-groups of size s as $\Omega(n\sqrt{\log s})$. Thus our experiments suggest that for randomly generated sub-groups, on average, the overhead is much less than in the worst case. The behavior of synchronization (see Figure 4) overhead is similar to the splitting overhead but the absolute value is smaller.

The results for experiment set 3 listed above are given in Figure 5. This demon-

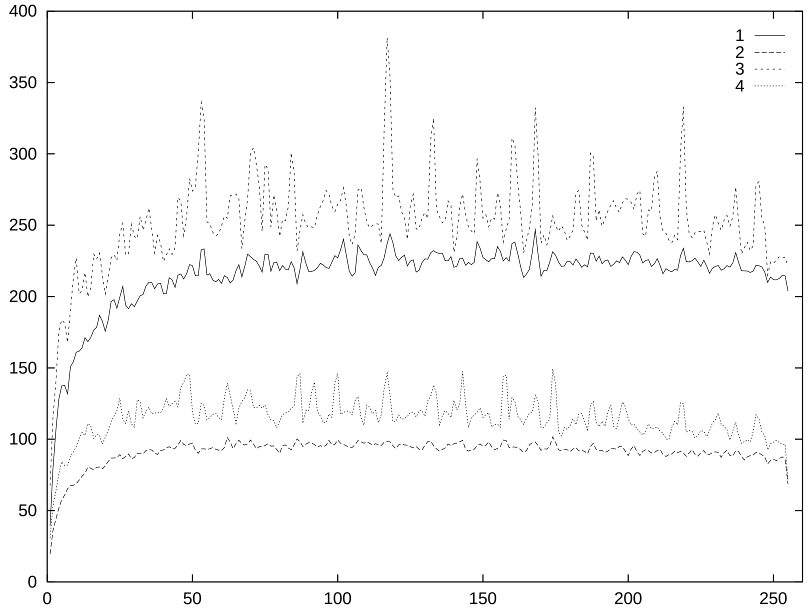

Figure 4: *Sub-group identification overhead for random subgroup vs. subgroup size (Hilbert indexing scheme): 1 – average, 3 – maximal (observed); synchronization overhead: 2 – average, 4 – maximal.*

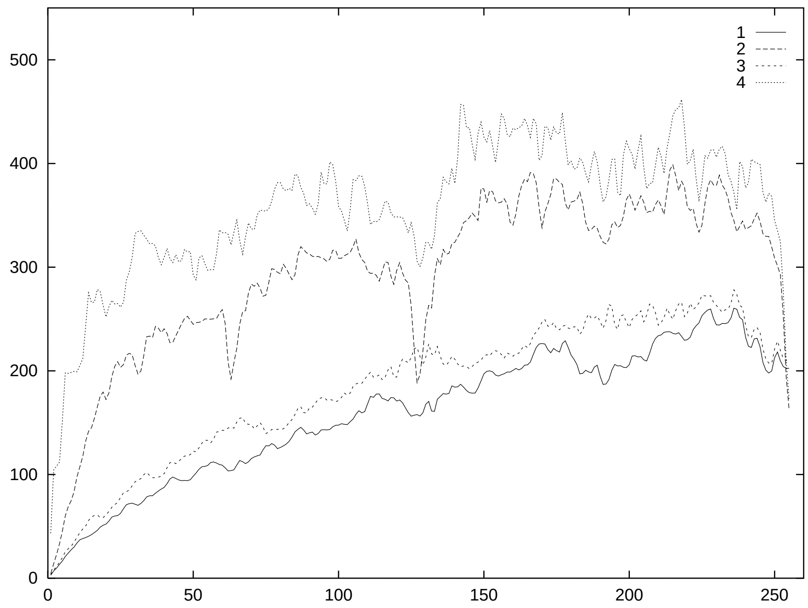

Figure 5: *Sub-group identification overhead for random subgroups with consecutively indexed nodes (Hilbert indexing) vs. subgroup size: 1 – average, 3 – maximal (observed); same for row major indexing: 2 – average, 4 – maximal.*

strates the influence of the locality preserved by Hilbert indexing on reducing the overhead. One may observe that the row major indexing scheme exhibits almost twice as large an overhead as than the Hilbert indexing scheme.

6 Related work

The SB-PRAM project [1] aims to implement a fairly direct realization of the PRAM model, in which 128 processors are connected via a butterfly network to 128 memory modules and lock-step synchronization is enforced at the level of machine instructions (i.e. in traditional PRAM style, across the whole machine, irrespective of higher level semantic activity). At this low level, synchronization is achieved through a development of the approach originally proposed by Ranade [11], with ghost packets and packet combination at network switches. With such a strong foundation (which we would, however, argue to be unscalable), higher level synchronization can be achieved in a straightforward way. Essentially the idea is that at each partition point, a new shared location (effectively a semaphore) is created for each new group. This is initialized (from 0) by an atomic increment from each group member. Synchronization is then achieved by requiring each group member to decrement the variable, busy-waiting until it is 0 once more. The variable is re-initialized and the synchronized processors proceed. Sequentialized access to the variable is avoided by combining in the network with what is effectively a standard 'fetch-and-add' operation. Notice that this method does not require the processors in a group to know the size of the group or indeed to know anything about the other members. In [9] it is reported that synchronization can be achieved in 10 SB-PRAM cycles, irrespective of the number of processors.

We could certainly adopt a scheme similar in principle (i.e. working at the level of emulated PRAM instructions). The main drawback would be that the combining techniques which make the method practical in the SB-PRAM for relatively small networks leads to $\Omega(n^2)$ worst-case synchronization overhead. Our tree-based synchronization is $O(n \log n)$ with high probability on meshes.

References

[1] Abolhassan, F., Drefenstedt, R., Keller, J., Paul, W.J., and Scheerer, D., "On the Physical Design of PRAMs", *Computer Journal*, Vol. 36, No 8, pp. 756-762, 1993.

[2] Bilardi, G., and Preparata, F.P., "Horizons of Parallel computation." in: A. Bensoussam, J.-P. Verjus (eds.), *Future Tendencies in Computer Science, Control and Applied Mathematics, Int. Conf. on the Occasion of the 25th Anniversary of INRIA*, LNCS 653, 1992, pp. 155-174.

[3] Blelloch, G.E., Hardwick, J.C., Sipelstein, J. and Zagha, M., "NESL User's Manual", Tech. Report CMU-CS-95-169, School of CS, Science, CMU, 1995.

[4] Chochia, G., Cole, M., and Heywood, T., "Implementing the Hierarchical PRAM on the 2D Mesh: Analyses and Experiments", *Proc. of the* 7^{th} *IEEE Symp. on Par. and Dist. Processing*, San Antonio, 1995, pp. 587-595.

[5] Hagerupt, T., Schmitt, A., and Seidel, H., "FORK: A high-level language for PRAMs", *Future Generation Computer Systems*, 8, 1992, pp. 379-393.

[6] Heywood, T., and Ranka, S., "A Practical Hierarchical Model of Parallel Computation. I. The Model", *Journal of Parallel and Distributed Computation*, 16, 1992, pp. 212-232.

[7] JáJá, J., "An Introduction to Parallel Algorithms", Addison-Wesley, 1992.

[8] Kaklamanis, C., and Persiano, G., "Branch-and Bound and Backtrack Search on Mesh-Connected Architectures", *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, 1992, pp. 118-126.

[9] Kessler C.W. and Seidl H., "Fork95 Language and Compiler for the SB-PRAM", *5th Intl. Workshop on Compilers for Parallel Computers"*, 1995.

[10] Leighton, T., Maggs, B, Ranade, A., and Rao, S., "Randomized Routing and Sorting on Fixed-Connected Networks", J. of Algs, July 1994.

[11] Ranade, A.G.,"How to Emulate Shared Memory", *Journal of Computer and System Sciences*, Vol. 42, 1991, pp. 307-326.

[12] Rajasekaran, S., and Tsantilas, T., "Optimal Routing Algorithms for Mesh-Connected Processor Arrays", *Algorithmica*, Vol. 8, 1992, pp. 21-38.

[13] Rajasekaran, S., *Personal communication.*

7 Appendix

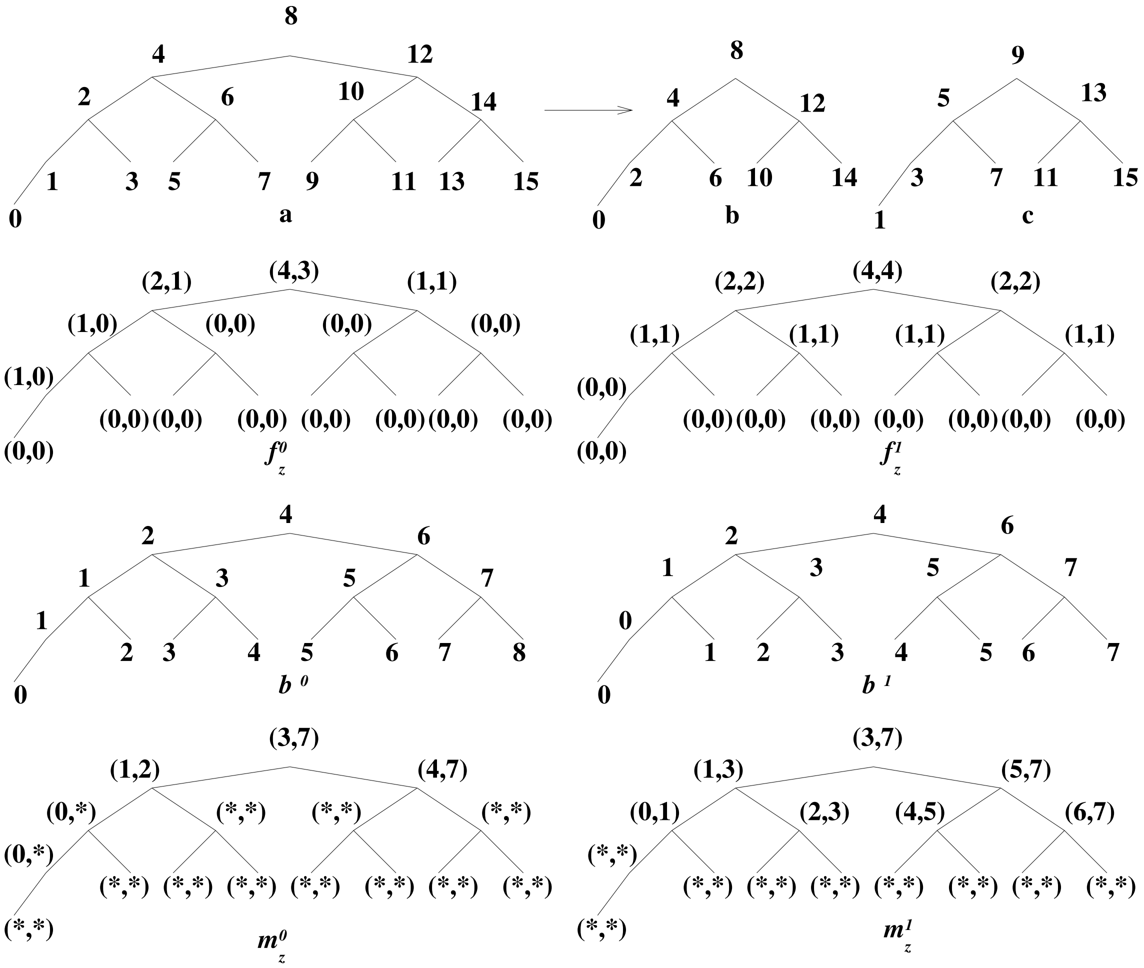

Figure 6: *Original sub-group with odd nodes in state 0, even nodes in state 1 and its synchronization tree: figure* ***a****. The sub-group split into two 0/1-subgroups and their synchronization trees: figures* ***b*** *and* ***c****. Functions f_z^x evaluated at the forward pass, functions b^x and m_z^x evaluated at the backward pass ($-\infty$ is shown as asterisks).*