Data Gathering on Reconfigurable Networks

by

Y. Ben-Asher and A. Schuster

Technical Report #696
October 1991
Data Gathering on Reconfigurable Networks

Yosi Ben-Asher∗ Assaf Schuster†

October 21, 1991

Abstract

Reconfigurable networks attract increased attention recently, as an extremely strong parallel model which is realizable in hardware. In this work we consider the basic problem of gathering information which is dispersed among the nodes of the network. We analyze the complexity of the problem on reconfigurable linear-arrays. The analysis introduces a novel criteria for the efficiency of reconfigurable network algorithms, namely the Bus-Usage. The Bus-Usage quantity measures the utilization of the network sub-buses by the algorithm. It is shown how this yields bounds on the algorithm run-time, by deriving a run-time to bus-usage trade-off.

∗Mathematics and computer science department, Haifa university
†Computer science department, Technion, Technion City, Haifa ISRAEL 32000. E-mail: assaf@cs.technion.ac.il
1 Introduction

Reconfigurable Networks (RN) attract increased attention recently, as an extremely fast parallel model. This increased interest is due to technological developments that have made it ready for current and future implementations.

The basic idea of a reconfigurable network is to enable flexible connection patterns, by allowing nodes to connect and disconnect their adjacent edges in various patterns. This yields a variety of possible topologies for the network, and enables the program to exploit this topological variety in order to speed up the computation.

Informally, a reconfigurable network operates as follows. Essentially, the edges of the network are viewed as building blocks for larger bus components. The network dynamically reconfigures itself at each time step, where an allowable configuration is a partition of the network into paths, or a set of edge-disjoint buses. A crucial point is that the reconfiguration process is carried out locally at each processor (or switch) of the network. That is, at the beginning of each step during the execution of a program on the RN, each switch of the network fixes its local configuration by partitioning its collection of edges into any combination of pairs and singletons. Two adjacent edges grouped by a switch into the same pair are viewed as (hardware) connected, and the processor may choose to listen to any incoming or passing message. The local partitioning may be taken by the switch oblivious to the configuration decision taken by the rest of the processors.

1.1 History and Realizability of the Model

The basic assumption concerning the behavior of the reconfigurable model (and indeed, of any other bus model) is that in any configuration, the time it takes to transmit along any bus is constant, regardless of the bus length. Obviously, this assumption is somewhat unrealistic. However, an important property of the reconfigurable networks model is the lack of diameter considerations. This advantage calls for underlying network topologies which are simple and easy to build, e.g., linear arrays, meshes and cubes (three dimensional arrays). Simple configurations, in turn, imply short wires and simple wiring (typically, machines based on these topologies can be embedded so that every edge has a constant wiring length which shorten the cycle of the machine and make the fixed bus transmission assumption acceptable, even for massively parallel machines.

Several authors considered the abstract reconfigurable model in the past. Nakatani [Nak87] considered comparison-based operations like merging, sorting and selection on reconfigurable arrays. Miller, Stout, Reisis and Kumar [MPRS87] and Reisis and Kumar [RP87] considered parallel computations and data movement operations on the reconfigurable mesh. Wang and Chen et. al. presented constant time algorithms for RNs, summarized in [Wan91]. Other algorithmic works can be found in the recent literature [EW91, JS91, WCL91]. In a series of papers, originating back to 1974, Rothstein et. al. investigated the the Bus Automata which is strongly related to the RN model [Rot76, Rot77, Rot88]. Moshel and Rothstein [MR79] showed that large classes of problems are solved in constant time on the BA. Many authors have considered the fault-tolerant properties of reconfiguring networks (see e.g. [MKS89]).

Early implementations, e.g. the CHiP project [Sny82] or the MP/C [AG82], focused on the...
construction of a globally controlled reconfigurable machines. Other experimental and commercial reconfigurable chips currently exist, consisting on thousands of switches and processors supported by a dynamically reconfiguring interconnection networks. Maresca and Li implemented a VLSI chip called YUPPIE (Yorktown Ultra-Parallel Polymorphic Image Engine) on a reconfigurable torus [LM89a, LM89b]. Another existing VLSI design is the CAL Chip, currently consisting of a 16 x 16 reconfigurable mesh of switches [GK89].

1.2 The Power of Reconfiguration

Reconfigurable Networks outperform PRAM's in the computation of several natural problems. For example, consider the computation of the parity function (or summation mod 2). The parity of $N^{1/2}$ inputs (given each at a different switch) is computed in $O(1)$ steps by a $N^{1/2} \times N^{1/2}$ reconfigurable mesh (see e.g. [BPRS91]), compared to the lower bound of $\Omega(\frac{\log N}{\log \log N})$ steps for CRCW PRAM having any polynomial number of processors [BH87]. In [BPRS91] it is shown how to sort $N$ keys in constant time on a reconfigurable construction, using $N^{1+\epsilon}$ switches and optimal layout area. In general, RN's seem to run $\log N$ times faster than EREW PRAM's and $\frac{\log N}{\log \log N}$ times faster than CRCW PRAM's. In [BPRS91] this gap is generally established. We refer the reader to [BPRS91, BPS91, Sch91] for other upper and lower bounds on the computational power of RN's.

A model that should be compared with the RN model is the (fixed topology) mesh with multiple buses, e.g. [Sto83, Sto86, BP91]. This model is based on a regular mesh augmented with a collection of buses providing broadcast facilities along every row and every column. It is important to note that multiple-bus models use the same assumptions as discussed above for RN's. Namely, multiple-bus models assume both broadcast capability and that a message may travel along any bus in a single step, where buses are of unbounded length. In spite of this fact, the gap between the two models is quite large: for example, in [BPRS91] it is shown how to compute parity of $N$ inputs in constant time using a simple reconfigurable construction with only $4N$ switches. This result should be compared with the $\Omega(N^{1/6})$ steps required to compute this function on the square mesh with multiple buses [PR87], and $\Omega(N^{1/8})$ steps needed for computing parity on any rectangular $N$-processor mesh with multiple buses [BP91].

1.3 This Work

In this work we discuss the power of the RN model in certain basic operations. We investigate operations that are very common and are frequently performed on parallel machines, giving upper and lower bounds for their complexity.

The most simple underlying network topology, is the reconfigurable linear-array (RLA). Reconfigurability implies that linear arrays, are formed as sub-networks of many other topologies. In particular, results for algorithms on linear arrays serve as upper bounds for other constructions containing them. Thus, this work is naturally focused on the investigation of RLA's, keeping in mind that the upper bounds given hold for many other constructions. It also turns out that the methods developed for achieving lower bounds for RLA's seem suitable for application on general topologies.

Although simple, the RLA of $N$ processors is a very strong network compared to other models. For
example, it computes the sum of $m \leq N$ inputs, arbitrarily dispersed among its processors, in $O(\log m)$ steps with high probability, or deterministically in $O(\log m \log^* N)$ steps [Sch91]. Such results are impossible using fixed-topology networks. As another example: using a well known technique, called “bus-splitting”, the RLA is capable of computing the OR of $N$ Boolean inputs in a single step, whereas the CREW PRAM, having any polynomial number of processors, requires $\Omega(\log N)$ steps [CDR86] (it is assumed that the input bits are given one at each processor).

One of the most basic operations of a parallel computer is to assemble data which is dispersed among the processors, into the buffers of a single processor. We call this operation data gathering. The inputs to the data gathering operation are usually intermediate results of previous, local computations taken by each individual processor. After the parallel work by the processors has terminated, the computed partial results need to be processed together into a global, final result, or assembled to produce the input for the next parallel section. Other cases involving data gathering are all the situations in which processors need to cooperate. For example, whenever some barrier synchronization mechanism is activated.

Sometimes data gathering involves concatenation of the data components, saving all the input bits. In many other cases, however, two partial results may be combined into a single, new partial result. Such situations are often called data reduction, since two partial results, once met, are combined into one by some reduction operation. We formulate this situation by assuming that every reduction operation belongs to the wide class of semi-group operations. In this context, data gathering is called global-product of semi-group elements that are initially stored at the processors. Combining two partial results into one requires a single application of the semi-group operation. This formulation is, on the one hand, restrictive, as our results apply for any arbitrary reduction operation (even a random one). On the other hand, it includes many important and natural problems such as parity, max, sum, etc.

In many situations, several data gathering operations are to be executed concurrently. Initially each processor contains $k$ data elements to be assembled with other data elements stored at the other processors. $k$ data gathering operations are performed, so that each data element participates in just one of them. The final results are stored arbitrarily at any of the network processors. This operation, called multiple-product, is formulated by taking $k$ global-products.

The multiple-product problem is a typical situation to appear on parallel machines. For example, consider programs that are highly parallelizable, so that they are compiled into $v$ virtual processes, to be run on a parallel machine having $p$ processors, where $p \ll v$. The parallel slackness concept, considered by several authors, is defined as the amount of $k = v/p$ processes that are assigned to, and executed by each processor during each “super step” of the parallel machine. Recently, Valiant showed that a certain amount of slackness is necessary and sufficient for optimal simulations of ideal shared memory models by realizable machines [Val90]. For schemes that use slackness, there will usually be $k$ partial results stored by each processor. Often the different results at a certain processor belong to different computations. Thus, at certain points during execution, $k$ different operations of data gathering are performed concurrently by the parallel machine.

In this paper we investigate the complexity of data gathering operations on reconfigurable networks, using the formulation of semi-group operations. The rest of the paper is as follows: section 1.3.1 gives the (informal) statement of our results. In section 2 we give the definition of the reconfigurable networks model. In section 3 the semi-Euler path is introduced and an upper bound for data gathering.
on RLA's is shown. In section 4 and the subsequent sections, the Bus-Usage technique is developed, yielding a matching lower bound for the data gathering problem.

1.3.1 Results

A well known algorithm computes any global-product on an N-processors RLA in \( \log N \) steps. Iterating this algorithm, it is straightforward to compute \( k \) global-products in \( O(k \log N) \) steps. Using pipeline, however, \( k \) global-products are computed by an \( N \)-processors RLA in \( O(k \log N) \) parallel steps. The algorithms given are generic, in the sense that the product may assume any semi-group operation, provided that this operation takes constant number of steps to compute on a single processor.

We define the concept of semi-Euler path capturing the existence of a linear array in a general graph. Explicit characterization of graphs containing a semi-Euler path (by means of their connectivity) is given, thus applying the upper bound for these underlying topologies, too. In this way it is shown that any upper bound for RLA's holds also for large classes of non-linear RN topologies, e.g. all graphs having even node degree and all graphs having connectivity not less than 4.

Our main result is a lower bound establishing the optimality of the above upper bound: \( k \) global-products require \( \Omega(k \log N) \) parallel steps to compute on an \( N \)-processors RLA. No restriction on the processor's instruction set is assumed, so that the lower bound is inherently related to the communication bottlenecks and the use of the sub-buses. We do, however, make two relatively natural assumptions for the way the algorithm operates, see section 4.

For the proof of the lower bound, a new technique is developed for estimating the efficiency of RN algorithms. The method measures the bus-usage, i.e. the amount of usage of sub-buses by the algorithm. The bus-usage method is important by itself and may provide further results for RN's analysis: on the one hand, if the bus-usage is small relative to the total bus length available, then it should be possible to accelerate the algorithm which does not utilize the communication edges. On the other hand, if the bus-usage that is required is proven to be very high, then it implies a lower bound for the algorithm running time. For the global-product operation, using a special technique for counting active communication edges the bus-usage is shown to trade for time, thus implying the lower bound.

2 The Model

A reconfigurable network (RN) operates in the single-instruction multiple-data (SIMD) mode. That is, the processors residing at the nodes of the network perform the same program synchronously, taking local decisions and calculations according to the input and locally stored data. A single node of the network may consist of a computing unit, a small buffer and a switch with reconnection capability. The buffer holds either an input item or a partial result. In the sequel, we interchange the notions of switch, processor and a network node.

A single time step of an RN computation is composed of the following substeps.

Substep 1: The network selects a configuration \( H \) of the buses, and reconfigures itself to \( H \). This is
done by local decisions taken at each switch individually.

**Substep 2:** One or more of the processors connected by a bus transmit a message on the bus. These processors are called the *speakers* of the bus. The message length is bounded by the *bandwidth* parameter of the network.

**Substep 3:** Some of the processors connected by the bus read the message transmitted on the bus by the speaker(s). These processors are referred to as the *readers* of the bus.

**Substep 4:** A constant time local computation is taken by every processor.

At each time step, a bus may take one of the following three states. *Idle:* no processor transmits, *Speak:* there is a single speaker, *Error:* there is more than one speaker. An Error state, reflecting a collision of several speakers, is detectable by all processors connected by the corresponding bus, but the messages are assumed to be destroyed.

The *bus-bandwidth* is the maximum size of a message transmitted on the bus at some time step. We assume that the bandwidth is sufficient for the transmission of a single data item or a single partial result.

The most simple connected topology is the Reconfigurable Linear Array (RLA). An RLA consists of $N$ processors labeled $\{1, \ldots, N\}$. Processor $i$ is connected to processors $i - 1$ and $i + 1$ if $2 \leq i \leq N - 1$, to processor 2 if $i = 1$ and to processor $N - 1$ if $i = N$. Thus, for a switch of an RLA there are only two local configurations: either connect or disconnect the edges. The set of possible global configurations of an RLA is the set of all $2^{N-2}$ splittings of the $N - 1$ edges into smaller linear arrays.

Let $G$ be a semi-group having an operator "$\cdot$". We call *global-product* the operation of computing

$$\psi(I_1, \cdots, I_N) = I_1 \cdot I_2 \cdot \cdots \cdot I_N$$

where $I_i$ denotes the element from $G$ which is stored at processor $i$ of the network. Similarly, $I_{i_1} \cdot I_{i_2} \cdot \cdots \cdot I_{i_M}$ for some sequence of indices $i_j, j = 1, 2, \cdots M$ is called a *partial-product*. We call *multiple-product* the operation of computing several $(k)$ global-products concurrently. Each set of $N$ inputs, for which the global-product is taken is called *input-set*. We assume that each set is assigned with a label, it's group-id, so that initially each processor contains $k$ labeled inputs, one for each input-set. The labels are taken from $\{1, \ldots, k\}$. Input-set (labeled) $j$ is taken from a semi-group $G^{(j)}$ having the operation $\cdot^{(j)}$. The eventual results of the global-product or the multiple-product operations may reside *arbitrarily* at any of the processors.

### 3 The Semi-Euler Path

**Proposition 3.1** *Global-product is computed in \(\log N\) steps on the \(N\)-nodes RLA.*

**Proof:** Basically, at the $i$th step, the RLA is split into non-intersecting sub-buses of length $2^i$, each containing a pair of processors storing partial results. One of these processors sends the partial result it holds on the sub-bus and the other one computes and stores the product.

Note that for certain operations we have a much better upper bound:

5
Proposition 3.2 [WC90] The RLA computes the OR (AND) of its inputs (given one bit at each processor) in a single step.

Definition 3.1 A semi-Euler path \( P \) in a graph \( G \) is a path visiting all nodes of \( G \), while passing at most once at any one edge. We say that a graph is semi-Eulerian if it contains a semi-Euler path. A reconfigurable network is semi-Eulerian if its underlying topology consists of a bounded-degree semi-Eulerian graph.

Corollary 3.1 Global-product is computed on any semi-Eulerian \( N \)-nodes \( RN \) in \( O(\log N) \) steps.

Proof: Let \( P \) be a semi-Euler path in \( R \), now use the algorithm of proposition 3.1. Note, however, that a semi-Euler path might "pass" up to \( d/2 \) times via a single node \( x \), where \( d \) is the number of edges connected to \( x \). Thus \( P \) must be connected as follows: let \( \{(i_1, o_1), \ldots, (i_m, o_m)\} \) be the \( m \) \( (\leq d/2) \) pairs of edges that are incident to \( x \) and which belong to \( P \). \( P \) "enters" \( x \) via the edge \( i_j \) and "leaves" it via \( o_j \). \( x \) arbitrarily chooses one of these pairs as his input-output (left-right) connections to the path \( R \), and fixes all other pairs to be connected throughout the algorithm execution.

In general, we have

Corollary 3.2 Results for \( dN/2 \)-nodes RLA's hold as upper-bounds for any \( N \)-node semi-Eulerian \( RN \) with maximum degree \( d \).

Which networks are semi-Eulerian? We first give a characterization of semi-Euler graphs.

Proposition 3.3 A graph is semi-Eulerian if and only if it has a sub-graph which is

(i) spanning,

(ii) connected,

(iii) no more than 2 of its nodes have odd degree.

Proof: From the definition, a connected graph is semi-Eulerian if and only if it has a spanning sub-graph which is Eulerian. (i) through (iii) is a characterization of Eulerian graphs.

Theorem 3.1 [New] A graph, \( G = (V, E) \), having edge - connectivity \( k \) is semi-Eulerian.

Proof: A \( k \) - edge - connected graph contains \( [K/2] \) edge - disjoint spanning trees (\([Nas61]\)). Hence, \( G \) contains 2 such trees, \( T_1 \) and \( T_2 \). Consider the sub-graph \( \hat{G} = (V, E - E(T_1)) \), where \( E(T_1) \) denote the edges of the tree \( T_1 \). \( \hat{G} \) is connected, so there exists a loop-free path \( PATH_{x,y} \) between any pair of nodes \( (x,y) \).

Denote \( V_{odd} \) the set of nodes having odd degree in \( G \). Split \( V_{odd} \) into pairs, and let \( PATH_{odd} \) denote (any) set of paths in \( \hat{G} \) connecting these pairs. Let \( E_{odd} \) denote the set of edges in \( E(\hat{G}) \) having an odd number of paths from \( PATH_{odd} \) traversing them. Let

\[
G' = (V, E(\hat{G}) - E_{odd} + E(T_1)) = (V, E - E_{odd})
\]
It is easy to verify that $G'$ is a subgraph of $G$ having all three properties from the characterization given by proposition 3.3.

The claim is not true for edge-connectivity 3, as is shown in an example by N. Linial: take the complete graph on 4 vertices $K_4$ as a base. Replace each vertex by the modified Petersen graph, which lacks a node, with 3 out-edges, as illustrated in Figure 1. If the resulted graph contains a semi-Euler path, then at least two of the modified Petersen graphs have a semi-Eulerian path starting and ending at the out-edges. Such a path cannot visit a node more than once, and can be completed as a Hamiltonian tour of the original Petersen graph, which is known not to exist.

Corollary 3.3 Multiple-product of $k$ input-sets is computed in $O(k \log N)$ steps on any semi-Eulerian $RN$.

Theorem 3.2 Multiple-product of $k$ input-sets is computed in $O(k \log N)$ on an RLA (and hence on any semi-Eulerian $RN$).

Proof: Assume for simplicity, that $N$ is a power of $k$. We sketch the algorithm informally.

Algorithm MULTIPROD:

The algorithm incorporates $\log_2 N$ phases. During the first phase the RLA is split into $\frac{N}{k}$ segments, each consisting of $k$ consecutive processors. All segments follow the same process: A "shift and add"-like move of the partial results from (say) left to right. After $2(k - 1)$ steps (partial) products for all input-sets have already arrived at processor $k$ of each segment, and the first phase is completed. The same process is repeated at the next phases, where only processors holding partial results perform additions while the others connect their left and right edges to form a contiguous bus.

The first phase is an evaluation of $\frac{N}{k}$ partial results of the global result. This is done for all input-sets. The second phase evaluates (for each input-set) $\frac{N}{k}$ partial results of the partial results which are the output of the first phase. After $\log_2 N$ such phases, processor $N$ contains all $k$ (final) results for the input-sets.
4 The Bus Usage Measure and a Lower Bound for Multiple-product

In the following sections we prove that the upper bound given by the algorithm for computing $k$ global-products is tight, for a certain "natural" class of algorithms (to which, for example, the algorithms given in section 3 belong). The algorithms in this class have no restriction on the processor's instruction set (for each global-product taken), so that the tradeoff involved is inherently related to the communication and the use of the sub-buses.

In the course of the proof, we show that the algorithms which follow a specific tree-like computation pattern are the most economic in using the bus. This is achieved using the definition of a new measure for the efficiency of an algorithm. The new quantity, called the bus-usage, captures the use of communication links as a computation resource. For the computation of global-product we prove a tradeoff for time vs. bus-usage.

In proving the lower bound, we assume that the binary operations that are involved in the global-product are commutative. This property is thus shown not to speed up computation on the RLA.

No mixing: one assumption made, namely the no mixing assumption, is that values issued on a bus always depend on a single input-set. This is a natural assumption when the transmitted values are partial results, and these may have bit representation as high as the bandwidth. Also, the situation (appearing so frequently in the literature) in which the processor's instruction set is limited to the semi-group operation, is a strengthened version of this assumption.

Residual transmission: we also assume that some "residual" information from each single input is transmitted all the way through partial results and messages to the final product (i.e., the computation is not purely "configurational computation" [Wan91]). This assumption is necessary and makes sense when single inputs are long enough. The lower bound does not hold in cases where single inputs are short. For example, the AND function on single bits is computed by an RLA in a single step (proposition 3.2), so $k$ global-products with the AND operation are computed in $k$ steps.

The residual transmission is a strong assumption. Its formal statement is given before the proof of lemma 5.1, where it is used. Here we explain the reasons and the justification for making it, and give some evidence that it holds. The assumption of residual transmission is related to the topology of the network, to the number of different local configurations which may be assumed by the switches, and to the size of the inputs. A stored value may cause certain sequences of local configurations, even if it is not actually transmitted (this is the intuitive notion of configurational computation as used by [Wan91]). However, when the product is taken over arguments from a large domain (say, each input argument is represented by more than $r$ bits, where $r$ is the algorithm run-time) and the number of local configurations of a switch is constant, then for an algorithm that violates the assumption we can always "fool" the network by finding two different input configurations giving the same global-product. Thus, if the algorithm is generic (suited for all semi-group operations) and does not depend on the size of the input, then it could not use reconfiguration alone to communicate data.

Finally, we assume a "weak" model: in one step a processor can read from only one of its inputs. This does not disturb the asymptotics of the lower bound, since (for the RLA) any $T$ steps computation by the strong model can be emulated by a $2T$ steps computation in the weak model.

**Theorem 4.1** Let $T$ denote the number of parallel steps required for the concurrent computation of $k$
global-products on the RLA. Then, under the stated assumptions on the algorithm, \( T = \Omega(k \log N) \).

We first define the bus-usage quantity: Let \( e_s = (s, s+1) \) be some edge of the RLA. Let \( 1 \leq t \leq T \) be a time step of a \( T \) steps program for computing \( k \) global-products on the RLA.

**Definition 4.1** The pair \( (e_s, t) \) is called a bus-unit for a computation of that program, if \( e_s \) belongs to one of the sub-buses of the configuration chosen at time \( t \) of the computation. The number of bus-units during a computation, i.e. the sum of all bus lengths chosen at the configurations during that computation, is called the bus-usage.

**Lemma 4.1** Even in the most economic situation, computing global-product in \( T \) steps involves a bus-usage of \( \frac{N(J+1)}{4} \), where \( J \) is the maximal positive integer such that \( J \leq \frac{\log N}{\log e_k} \).

**Proof of Theorem 4.1:** Given lemma 4.1, putting \( T \leq 2k \log N \) (see corollary 3.3) and iterating, we get that \( J \geq \log N/(\log 4k + \log \log 4k + \log \log \log 4k + \ldots ) \). The no mixing assumption implies that the total usage of the sub-buses is the sum of the usage made by the individual products. Hence the total bus-usage during a \( T \) steps computation of \( k \) global-products is at least \( CNk \frac{\log N}{\log e_k} \), for some constant \( C \). Since the maximum bus-usage at any one step is \( N \) the claim follows. 

For the proof of lemma 4.1 which is given in section 7, we need some more definitions and preparation work. We first prove that global-product computations which are most economic in bus-usage are always found in a restricted and relatively "simple" family of algorithms.

**5 Single-Reader Tree - Computations**

**Definition 5.1** A computation for which any partial result that is computed (or sent on a bus) is a partial-product, and for which any partial-product that is computed takes part in the computation of a single partial-product at a later step is called a tree-computation. A computation having a single reader at any sub-bus of any configuration is called a single-reader-computation.

**Lemma 5.1** There is a single-reader tree-computation of global-product achieving the minimal bus-usage among all \( T \) steps global-product computations.

Let \( A \) be a program for computing global-product and let \( D \) denote a \( T \) steps computation of \( A \). We define \( \text{DAG}_D \) describing \( D \) and its communication pattern. The nodes of \( \text{DAG}_D \) are all the pairs \( (p = \text{processor}, t = \text{time}), \) for \( 1 \leq p \leq N \) and \( 0 \leq t \leq T \). The nodes \( (p, 0) \) are called inputs. There are two kinds of edges in the DAG:

**Communication edges:** \( (p, t) \rightarrow (q, t) \) - standing for a message issued by processor \( p \) on a bus at time \( t \) and was read by processor \( q \).
Figure 2: The DAG representation of 4 inputs global-product computation: \((p, t) = (\text{processor}, \text{time})\), horizontal edges stand for transmission of messages on the bus.

**Time-Flow edges:** \((p, t) \rightarrow (p, t + 1)\) - for all \(1 \leq p \leq N\), \(0 \leq t < T\).

Node \((p, T)\), where \(p\) is the processor computing the result at time \(T\), is called the mroot. If there are several then any of them can be considered the mroot. We assume that the mroot is the first processor having the chance (i.e. sufficient information) of computing the result, so that this computation is not postponed to later steps.

All communication edges originating from a node \((p, t)\), stand for node \(p\) transmitting on a sub-bus at time \(t\). The destinations of the edges are its readers. Note that each sub-bus of a configuration, and the total bus-usage are represented this way. In particular, we have:

**Proposition 5.1** Any computation of global-product which is represented by a DAG having a partial set of\(D\)'s communication edges and the same set of nodes, has a bus-usage which is less than or equal to that of \(D\).

In figure 2, a DAG is illustrated, associated with a computation of the global-product of 4 inputs. Note processor 1 reading a message sent by processor 4 at time step 2, but it does not use this information. This demonstrate an inefficiency of the computation, as it "wastes" two bus-units.

We can now formalize the residual transmission assumption:

**Assumption 5.1** The DAG of any global-product computation contains a (directed) path from each minput to the mroot.

Using assumption 5.1, we define a new graph \(TREE_D\) as follows. For each minput choose a directed path to mroot. The choice of paths is taken one at a time: choose any path for the first minput; given a set of already chosen paths, choose the next path so that once it reaches a node of an already chosen
path, it follows the previously-chosen path route to the mroot. Finally, remove all edges of \(DAGD\) which do not take part in any of the chosen paths.

The remaining graph \(TREE_D\) is obviously a tree, having edges which are a partial set of the edges of \(DAGD\). It is a binary tree whose internal nodes have one communication edge and one time-flow edge leading to their sons. The leaves of the tree are the minputs and the mroot is its root. When two sets of paths enter the same node, one via the communication in-edge and the other via the time-flow in-edge, the two sets are disjoint. Hence we can realize an algorithm \(\hat{D}\) for which \(TREE_D\) represents its communication pattern: a set of paths originating at processors \(i_1, \ldots, i_k\) and passing through node \((p, t)\) stand for the partial-product \(I_{i_1} \cdots I_{i_k}\) being stored at processor \(p\) at time \(t\). Such a set moving via a communication edge stand for that partial-product being transmitted over the corresponding bus. When two such (disjoint) sets "meet" at node \((p, t)\), this means processor \(p\) computes the combined result of the two corresponding partial-products, after reading the one coming from the communication in-edge. Since \(TREE_D\) is a tree we also conclude that in the configurations chosen for \(\hat{D}\) there is only one reader at each sub-bus.

**Proof of lemma 5.1** We summarize the above discussion: \(\hat{D}\) is a \(T\) steps tree - computation of global-product having a single reader at each sub-bus and, by proposition 5.1, bus-usage no higher than that of \(D\). Thus single-reader \(T\)-steps tree - computations of global-product achieve the minimum bus-usage.

Having lemma 5.1 proven, we proceed in proving lemma 4.1, the bus-usage to time tradeoff, while considering single-reader tree - computations only. For such computations, the communication pattern DAG forms a tree. The rest of the proof is an analysis of the structure of such trees, for minimal bus-usage computations. Counting sub-trees we are able to bound from bellow the total number of bus-units that compose the communication edges of the tree, thus get a lower bound for the bus-usage.

### 6 The Peeling Process

Consider a collection \(U\) of all the bus-units during a tree - computation of global-product which minimizes the bus-usage. We define a process consisting of consecutive peeling steps.

**Peeling step** \(j\): For each \(i \in \{1, \ldots, N - 1\}\), if there exist bus-units \((e_i, t)\) in \(U\) (recall \(e_i = (i, i + 1)\) is an edge of the RLA), then remove from \(U\) the one with the largest value of \(t\).

\(\chi_j\) denotes the total number of removed bus-units at this peeling step.

Clearly,

\[
\text{bus - usage} = \sum_{j=1}^{T} \chi_j
\]

**Definition 6.1** We say that a tree (sub-tree) forks at peeling step \(j\), if one of the bus-units forming the communication edges connecting either of the mroots of his two sons to his own mroot is first removed from \(U\) at that step. The two sons (sub-trees) are said to be exposed at this step.
that a sub-tree is ripe when it is exposed but not forked. The height of a sub-tree having \((w, h)\) as its mroot is \(h\).

In particular, when an minput is exposed, it never forks, so it stays ripe until the termination of the peeling process. The peeling-process terminates when all minputs are exposed.

The number of ripe sub-trees at any peeling step bounds from above the number of exposed minputs at that step. In what follows we show that this bound implies a lower bound for \(X_j\), the number of peeled bus units at the \(j\)th peeling step.

In order to bound the number of ripe sub-trees we consider again the construction of \(TREE_D\) in the proof of lemma 5.1. We repeat a similar construction, only that this time we let each communication edge split to its composing bus-units, making each of them a directed edge. Observe that in the current tree construction there are no two different “branches” of the tree which share a common bus-unit. When a bus-unit is chosen as a part of some leaf-to-mroot path, then all paths chosen later join him, once met at the vertices of the DAG composing that bus-unit. Similarly, two different branches reach the same node of the DAG only if they are “combined” at this node. This property, called the no overlap property, does not hold for the original tree construction, since communication-edges may be composed of several bus-units.

The no overlap property implies that a sub-tree always computes the partial-product of all inputs from a certain segment of consecutive processors of the RLA. We conclude that even if the semi-group operation involved in the global-product is commutative, the computation cannot use this to reduce its usage of the bus. For the rest of the proof we assume that the no overlap property holds for all trees (and thus for all computations considered).

When considering the constructed trees, we say that a node of the DAG is an internal node of the tree only if it is of in degree 2. A sub-tree of the global tree is labeled by the path of internal nodes leading to its mroot from the global mroot, e.g. the vertex at the end of the path \(mroot \rightarrow Left \rightarrow Left \rightarrow Right \rightarrow Left\) is denoted \(T_{L^2RL}\). A sub-tree which is denoted by \(T_{Li} (T_{Ri})\) for some \(i\), is called the \(i\)th leftmost (rightmost) sub-tree. A sub-tree which is not leftmost nor rightmost is called internal.

**Proposition 6.1** All edges of an internal sub-tree are kept in \(U\) while peeling the global tree.

**Proof:** Let \(T\) be an internal sub-tree. By the definition of internal nodes and sub-trees, \(T\) has a sub-tree \(T^L\) to its left and a sub-tree \(T^R\) to its right computing partial-products of minputs to its left and to its right, correspondingly. The partial-products computed by \(T, T^L, T^R\) are disjoint. Let \(I^L\) be one of the minputs in \(T^L\) and \(I^R\) be one of the minputs in \(T^R\).

There are directed paths \(d^L\) and \(d^R\) in the (global) tree, leading from \(I^L\) and \(I^R\) (correspondingly) to the mroot. Let \(u = (e, t)\) be some arbitrary bus-unit, which is one of those composing the edges of \(T\). \(u\) does not take part in \(d^L\) or in \(d^R\), since by the construction of the tree, this would imply that the partial-product computed by \(T\) contains the input appearing at \(I^L\) or \(I^R\), correspondingly. Since \(d^L\) and \(d^R\) do “meet” on the way to the mroot, their union must contain a bus-unit \(u' = (e, t')\) where \(t' > t\). Thus, by the definition of the peeling process, \(u\) would not be peeled at the next peeling step. Since \(u\) was chosen arbitrarily from among the bus-units composing \(T\), the claim follows.
From Proposition 6.1 we conclude that

\textit{internal sub-trees never fork.}\hspace{1cm} (2)

Property (2) by itself implies a recursive relation for \(F(T, L)\), the number of ripe sub-trees after the \(L\)th peeling step. Seeking for a deeper and a rather intuitive understanding of the communication pattern of minimal bus-usage computations, we also get that

\textit{non internal sub-trees always fork.}\hspace{1cm} (3)

The proof of (3) is by induction on the depth of the sub-tree and consists of exhaustive consideration of several cases. It is given in section 8.

7 Proof of Time to Bus-usage Tradeoff

Using (2) and (3) we have

\textbf{Proposition 7.1} \hspace{0.5cm} \textit{Let} \(S\) \textit{be a sub-tree of height} \(h\) \textit{which is ripe at the beginning of peeling step} \(j\). \textit{Then, as a result of peeling step} \(j\), \textit{at most} \(2i - 2\) \textit{sub-trees of} \(S\) \textit{of height greater than or equal to} \(h - i\) \textit{are ripe for all} \(1 \leq i \leq h - 1\) \textit{and a total of at most} \(2h\) \textit{sub-trees of} \(S\) \textit{are ripe.}

With this proposition we have the exact composition of ripe sub-trees and exposed minputs that are left after the peeling step.

\textbf{Proof of Proposition 7.1}: The two sons \(T_L\) and \(T_R\) of the peeled tree, which are the computations of the two partial-products that are combined at the global mroot at height \(h\), are each of height \(\leq h - 1\). By (3), since they are leftmost and rightmost correspondingly, they fork. Consider \(T_L\) and its sons (considerations are equivalent for the “right side” of the tree): \(T_{LR}\) is internal and so it becomes ripe, it is of height at most \(h - 2\). \(T_L\) is a leftmost sub-tree and forks too.

In general, the same process occurs for \(T_L\) (which, by (3), forks as a leftmost tree): \(T_{LR}\) is internal and so it becomes ripe, it is of height at most \(h - i - 1\). \(T_{L+1}\) forks as a leftmost tree, unless it is of height zero, so it becomes an exposed minput.

Thus at most \(h\) sub-trees at the “left side” of the tree become ripe: at most 1 of height greater than or equal to \(h - 2\), at most 2 of height greater than or equal to \(h - 3\) etc.

Finally, we are able to conclude the proof of the time to bus-usage tradeoff:

\textbf{Proof of lemma 4.1}: Given proposition 7.1 it is obvious what construction (if exists) brings the number of ripe sub-trees at any step of the peeling-process to a maximum: each sub-tree of height \(h\), once being peeled, is inherited by \(2h - 4\) ripe sub-trees, 2 of height \(k\) for all \(1 \leq k \leq h - 2\), and 4 exposed minputs.

We get a recursion relation bounding \(F(T, L)\), the number of ripe sub-trees of a global tree of height \(T\) after the \(L\)th peeling step:

\[
F(T, L) \leq 4 + 2 \sum_{i=1}^{T-2} F(i, L - 1) \quad \text{where} \quad F(0, L) = 1 \quad F(T, 0) = 1 \quad F(1, L > 0) = 2
\] (4)
Observe that the process terminates after at most $T/2$ peeling steps, so $L \leq T/2$. Using this we get:

$$F(T, L) \leq 2 + 2 \sum_{i=1}^{T-1} F(i, L - 1) \quad \text{where} \quad F(0, L) = F(T, 0) = 1$$

$$= \sum_{k=1}^{L} 2^k \sum_{i_1}^{T-1} \sum_{i_2}^{i_1-1} \cdots \sum_{i_{k-1}}^{i_{k-2}-1} (1) + 2^L \sum_{i_1}^{T-1} \sum_{i_2}^{i_1-1} \cdots \sum_{i_{k-1}}^{i_{k-2}-1} (1)$$

$$= \sum_{i=1}^{L} 2^i \left( \frac{T-1}{i-1} \right) + 2^L \left( \frac{T-1}{L} \right) < 2^L \left( \frac{T}{L} \right) \leq \left( \frac{2eT}{L} \right)^L$$

Recall that the number of ripe sub-trees bounds the number of exposed minputs. Also, each unexposed minput has at least one bus-unit attached to it which is removed from $U$ at the next peeling step, but each bus-unit is attached to at most 2 unexposed minputs. Thus the number of bus-units being removed from $U$ at some step is at least half the number of unexposed processors at the end of the previous step. Thus

$$\chi_j \geq \frac{N - F(T, j - 1)}{2}$$

(5)

Recall that $J$ is the maximal integer such that $(2eT)^J \leq N/2$, so for all $0 \leq j \leq J$: $F(T, j) \leq \frac{N}{4}$ and $\chi_{j+1} \geq \frac{N}{4}$. Hence bus-usage = $\sum_{j=1}^{T} \chi_{j} \geq \frac{(J+1)N}{4}$.

8 Stairway Structure for Minimal Bus-Usage Computations

Proposition 8.1 Unless it is composed of a single processor, a leftmost (or, similarly, a rightmost) sub-tree of a peeled tree forks and its sons are exposed.

Let $RES_{\text{path}}$ and $p_{\text{path}}$ denote the result computed by $T_{\text{path}}$ and the processor by which it is first computed correspondingly, where path is a path from mroot of the global tree to mroot of $T_{\text{path}}$. In other words, if $T_{\text{path}}$ is a tree of height $h$ then $(p_{\text{path}}, h)$ is its mroot and $RES_{\text{path}}$ is the value computed by $p_{\text{path}}$ at time $h$. We refer to left side and right side of processors on the bus in the natural manner, and we assume that processor-ids are increasing left to right.

Basically for minimal bus-usage tree - computations, we show (by induction) that for each $i > 1$ for which $T_L_i$ is a subtree: $p_{L_{i-1}} = p_{L_{i-1}R}$. In other words $RES_{L_{i-1}} = RES_L \bullet RES_{L_{i-1}R}$ is computed at the same processor computing $RES_{L_{i-1}R}$. This property implies a "stairway" structure of the left side of the tree, as depicted in Figure 3: a tree-node $(p, t)$ is represented by the point $(p, t)$ on the 2-dimensional plane. The claim obviously follows.

For the case $i = 2$: $RES_{L_{i-2}}$ is the global result $RES$ and is constructed out of $RES_L$ and $RES_R$. The claim is that $RES_L$ is computed at $p_{LR}$.

Figure 4 shows the saving in bus-units that could have resulted from a slight change of the computation, had it been the case that $RES_L$ is computed at a processor (strictly) to the right of $p_{LR}$. Let us describe this situation in detail. If $p_L$ and $p_R$ are at edge-distance $d$ then $d$ bus-units must be spent for $RES_L$ and $RES_R$ to meet at the mroot. If $RES$ is computed to the left of $p_L$ or to the right
Figure 3: "Stairway" structure of a minimal bus-usage tree - computation

Figure 4: Saving bus units in case $p_L$ is to the right of $p_{LR}$

Figure 5: Saving bus units in case $p_L$ is to the left of $p_{LR}$
of $p_R$ then more then $d$ bus-units are spent, hence we may assume that this is not the case. Suppose $p_L$ is $l$ edges to the right of $p_{LR}$. Thus both $RES_{LL}$ and $RES_{LR}$ are moving through these $l$ edges at different time-steps. From the no overlap property of the computation tree we conclude that $RES_{LR}$ reaches $p_L$ before $RES_{LL}$, say at time-step $t_1$. Let $RES_{LL}$ reach $p_L$ at time-step $t_2$, when $RES_L$ is also computed by that processor.

We now augment the computation in a way which results in a gain of $l$ bus-units and no possible loss. We already saw that $RES_L$, if it is transmitted by $p_L$, then it moves to the right. In the augmented computation, $RES_{LR}$ remains in $p_{LR}$ after it is computed. In the original computation $RES_{LL}$ moves to $p_L$ and thus may be read by $p_{LR}$ at time $t_3 \leq t_2$. In the augmented computation $p_{LR}$ actually reads $RES_{LL}$ and computes $RES_L$, so by the time $t_3$ $p_{LR}$ holds $RES_L$. $RES_{LL}$ does not move to the right of $p_{LR} = p_L$. For the rest of the computation we consider two cases:

(i) At some step $t_4 > t_2$ of the original computation $RES_L$ is sent to the right from $p_L$. The same is done at $t_4$ of the augmented computation by attaching the additional $l$ edges between $p_{LR}$ and $p_L$ to the bus (note that they are not used at that step of the original computation).

(ii) $RES_R$ reaches $p_L$ at time $t_4 > t_2$ when $RES$ is computed by $p_L = mroot$. For the augmented computation the bus used for moving $RES_R$ at $t_4$ is made longer by using the $l$ edges between $p_L$ and $p_{LR}$.

In any case, the augmented computation uses $l$ bus-units less then the original.

Figure 5 illustrates a similar saving for the case that $p_L$ is strictly to the left of $p_{LR}$. For the inductive step we assume that $p_{Li-2} = p_{Li-2R}$ and show that $p_{Li-1} = p_{Li-1R}$. We omit the detailed discussion which is similar to the base case. Figure 6 illustrates the augmentation of the computation when $p_{Li-1}$ is to the right of $p_{Li-1R}$. Figure 7 shows the augmentation when $p_{Li-1}$ is to the left of $p_{Li-1R}$.  

References


Figure 7: Saving bus units in case $p_{L_{i-1}}$ is to the left of $p_{L_{i-1}R}$


