On the Dynamic Initialization of Parallel Computers *

Stephan Olariu† Ivan Stojmenović‡ Albert Y. Zomaya§

Abstract

The incremental and dynamic construction of interconnection networks from smaller components often leaves the fundamental problem of assigning addresses to processors to be contended with at power-up time. The problem is fundamental, for virtually all parallel algorithms known to the authors assume that the processors know their global coordinates within the newly created entity. We refer to this problem as the initialization problem. Rather surprisingly, the initialization problem has not received much attention in the literature. Our main contribution is to present parallel algorithms for the initialization problem on a number of network topologies, including complete binary trees, meshes of trees, pyramids, linear arrays, rings, meshes, tori, higher dimensional meshes and tori, hypercubes, butterflies, linear arrays with a global bus, rings with a global bus and meshes with multiple broadcasting, under various assumptions about edge labels, leader existence, and a priori knowledge of the number of nodes in the network. With two exceptions, the proposed algorithms are optimal. Keywords: parallel architectures, initialization, linear arrays, rings, binary trees, meshes, tori, meshes with broadcasting, hypercubes.

*Work supported by NATO grant CRG-950744, NSF grant CCR9522093, ONR grants N00014-95-1-0779 and N00014-97-1-0526, NSERC grant OGPIN007, and by Australian Research Council grant 04/15/412/131.
†Department of Computer Science, Old Dominion University, Norfolk, VA 23529, U.S.A.
‡Department of Computer Science, University of Ottawa, Ottawa, Ontario, K1N 9B4, Canada
§Department of Electrical and Electronic Engineering, University of Western Australia, Perth, WA 6007, Australia
1 Introduction and Motivation

In static interconnection networks the processors are typically aware of their location within the network. Their “address” is either permanently stored in some local memory or is received at start-up time from a master processor. However, when the network is dynamically constructed from smaller components, or when a portion of existing network is allocated for solving a task in a multitasking environment, the processors may not be automatically given their addresses in the newly constructed entity. Yet, virtually all the parallel algorithms proposed in the literature assume that the processors know their new addresses. Clearly, such an assumption is not always justified. The problem is compounded by the availability of powerful and flexible computing devices – for example, transputers – that makes the dynamic construction of various interconnection topologies a reality. Not surprisingly, the problem of building networks incrementally from small building blocks has received a good deal of attention in the literature [5, 17]. Once such a construction is complete, it is necessary to inform every processor of the newly created network about its identity within the network. Since, as mentioned, virtually all known parallel algorithms assume that every processor knows its own coordinates within the machine, this initialization must be done at power-up time.

Rather surprisingly, this fundamental problem has been largely overlooked in the literature [2, 10, 15]. Recently, Nakano [10] described time optimal initialization algorithms for reconfigurable meshes. Lipton and Park [8] presented a probabilistic protocol that solves the initialization problem for asynchronous processors that communicate through shared memory. The protocol is distributed requiring no central arbiter and symmetric in that all processors start in exactly the same state. Briefly, each of \( n \) processors generates a number between 1 and \( n^2 \) repeatedly until all generated numbers are distinct. It seems that the protocol can also be used in a parallel shared memory setting. Given a \( k \)-dimensional hypercube with \( n = 2^k \) nodes, Bhagavathi et al. [2] presented a simple greedy \( O(n \log n) \) time sequential algorithm that returns a labeling of the nodes with binary strings of length \( k \) such that two nodes are neighbors in the hypercube if, and only if, their labels differ in exactly one bit. Bhagavathi et al. [2] left open the problem of finding an efficient parallel initialization algorithm for the hypercube.

In this work while dealing with the initialization problem we assume that
processors in a dynamically constructed network know the type of interconnection network that they belong to: a tree, a mesh, a hypercube, etc. and that they will synchronously run a parallel algorithm (supplied, say, by the master processor) to determine their address within the new network. A related problem, not addressed here, is the recognition problem, in which the processors have to verify whether they belong to a specific kind of network (e.g. hypercube). Ramarao [14] described an unified distributed algorithm to recognize whether a given network is a tree, ring, star, complete graph, or a bipartite graph.

In this paper we state the initialization problem as an important task for any interconnection network model of parallel computation and solve it for a number of well known models. We consider several variants and assumptions about the newly constructed network. In each case, a solution under a more general assumption X1 is also solution to the special case X2. On the other hand, the assumption X2 may allow a faster solution. Here, X is one of the possible assumptions about edges (E), the existence of a leader (L), or a priori knowledge about the number of nodes (N) in the network.

One of the questions about the new network is whether edges\(^1\) are labeled in any way, i.e. whether they have a “sense of direction” (cf. [4]). For example, it is important to know whether left and right directions on a ring or linear array, north, south, east and west directions for a new mesh, and link numbers on a hypercube are known by each processor in the platform. It is also possible that, initially, the edges are not marked [2, 4, 8, 14]. For instance, the recognition algorithms in [2, 14] have such assumption. We consider two possible assumptions about edge labels in the network:

E1. Edges are not labeled or marked in any way.

E2. Edges of the network are labeled by link number or left, right, top, bottom etc. orientation.

Edges are assumed to be full duplex, that is, processors can simultaneously send and receive a message from a neighbor. This bidirectional communication is usually implemented via specially designated memory registers within each processor.

Since the processors are assumed to have designated local registers to receive messages from their neighbors, we assume that they have a concurrent

\(^1\)As usual, edges describe nearest-neighbor connections within the topology at hand.
read capability: that is, they are able to receive messages simultaneously from several neighbors. However, they may use only one of the received messages at any one time in their local computation.

There are two common assumptions in the literature about concurrent write capabilities of the processors in a network. In the one-port model, a processor may use only one of its links to send a message, at any one time, in other words a processor may communicate with only one neighbor in each time step. On the other hand, in the all-port model, each processor can send the same message on all its links simultaneously. All the algorithms presented in this paper assume the more restrictive one-port model. However, it may be the case that for some models, not addressed in this paper, the all-port model yields more efficient initialization algorithms than their one-port counterparts.

Processors of the new network are assumed to be physically identical. However, they need a procedure to choose a leader between any two neighbors. The leader election is made on the basis of a datum that is distinct for each processor. On the other hand, it is well known [1] that there is no deterministic algorithm for leader election in anonymous\(^2\) systems where the communication graph is very regular, e.g. ring or a clique. Such a distinct datum may become available in several ways. For instance, processors can be distinguished by pairwise distinct serial numbers provided at fabrication time and recorded in their local memory, or alternatively, by generating a random number yielding, with high probability, a unique number for each processor. We shall also assume that the processors can detect which of their ports are linked to a neighbor. Therefore, they can count the number of their neighbors. Border processors in asymmetric networks – for example, linear arrays, meshes, complete binary trees, are thus able to identify themselves as such. The leader election problem is studied extensively in literature, using mainly distributed computing models [1, 16]. It is clear that the initialization problem is more difficult than leader election; indeed, the leader election problem can be viewed as the problem of identifying a single processor. We now make two natural assumptions about the existence of a leading processor in the network:

L1. There is no leading processor in the network. Processors must run a

\(^{2}\text{I.e. the processors run an identical program and they do not have distinct integer identifiers}\)
leader-election algorithm to determine the leader.

L2. A leader has been designated by a host processor.

Finally, we make two assumptions about whether or not the number \( n \) of processors in the newly created entity is known a priory.

N1. The number of processors is not known in advance.

N2. The number of processors is known at the beginning of the initialization algorithm.

The main contribution of this work is to solve the initialization problem for:

- an \( n \)-node complete binary tree of processors under the assumptions E1, L1, and N1 in \( \Theta(\log n) \) time;
- an \( n \)-processor mesh of trees under the assumptions E1, L1, and N1 in \( \Theta(\log n) \) time;
- an \( n \)-processor pyramid under the assumptions E1, L1, and N1 in \( \Theta(\log n) \) time;
- an \( n \)-processor linear array or ring under the assumptions E1, L1, and N1 in \( \Theta(n) \) time;
- \( n \)-processor meshes and tori of size \( \sqrt{n} \times \sqrt{n} \) under the assumptions E1, L1, and N1 in \( \Theta(\sqrt{n}) \) time;
- an \( n \)-processor hypercube under the assumptions E1, L1, and N1 in \( O(\log^2 n) \) time;
- an \( n \)-processor hypercube under the assumptions E2, N1, and L1 in \( \Theta(\log n) \) time;
- an \( n \)-processor butterfly under the assumptions E1, N1, and L1 in \( \Theta(\log n) \) time;
- an \( n \)-processor linear array with a global bus under the assumptions N1 and either E2, L1 or E1, L2 in \( \Theta(\sqrt{n}) \) time; similarly, for an \( n \)-processor and ring with a global bus under assumptions L2, E1, and N1 in \( \Theta(\sqrt{n}) \) time;
• an $n$-processor mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$ in $O(n^{1/4})$ time under the same assumptions as for the linear array with a global bus.

2 Initializing Complete Binary Trees, Meshes of Trees, and Pyramids

We begin by discussing the task of initializing a complete binary tree under the very general assumptions E1, L1 and N1. To begin, the unique processor that detects the presence of exactly two neighbors declares itself the leader. Moreover, the tree will be rooted at the leader.

The basic idea of the initialization is that the processors learn about their identity upon receipt of a bit vector from their parent in the tree. Originally the root processor sets its own vector to 000..01. In the next step the root will send to its two neighbors its own bit vector along with 0 to one of the children, 1 to the other. It is clear that this establishes a left child and a right child of the root.

Upon receiving the bit vector from its parent, each processor shifts its contents left one position and copies in the rightmost position the bit sent by the parent. The process is repeated as described, with each processor passing on to its two children its own bit vector along with an extra bit identifying the child as left or right. It is further easy to see that the resulting labeling corresponds to a left-to-right breadth-first numbering of the tree with the root receiving label 1.

To argue about the time optimality, note that in the end, all bit vectors must be distinct and in the transition from one level to the next at most one bit is changed. We have thus proved the following result.

**Theorem 2.1.** Under the assumptions E1, L1, and N1, the task of initializing a complete binary tree of $n$ processors can be performed in $\Theta(\log n)$ time. $\Box$

A mesh of trees network (MOT, for short), introduced in [12], consists of a mesh connected computer with $m$ rows and $m$ columns augmented with a complete binary tree in each row and each column, such that the processors in a given row or column serve as leaves in the corresponding tree. For simplicity, we assume that $m$ is a power of two;
however, the initialization algorithm can be modified to run for any choice of $m$. We assume that there are $m - 1$ additional processors per row and column, which are internal nodes of augmented binary trees. Thus, the MOT consists of $n = 3m^2 - 2m$ nodes.

The roots of $2m$ binary trees are the only nodes with exactly two neighbors, and they will start the initialization process by sending their data down the trees, looking for leaves, which are first reached nodes with more than three neighbors. These leaves are linked into a mesh connected computer by edges which were not used in the leaf search process. The same process also establishes parent children relationships in all binary trees, identifies the levels of each node in the tree, and detects the size of network, i.e. $m$ and $n$ (by leaf nodes). Further, the two unique root data received by a leaf suffice to find (after exchanging the information among neighbors) one or two neighbors (and corresponding edges) belonging to the same tree, for each of two trees over a given leaf. Among the identified leaves, exactly four have precisely four neighbors, namely the corners of underlying mesh computer. The four corner processors run a leader election process as follows. They send their data up each of two trees to which they belong. Each root receives data from two associated corners and returns both of them to both corners. Next, each corner sends again both data to both associated roots, and these roots return all collected data back to their corners. The corner processors now have data from other corner processors and, consequently, are able to elect a leader. Moreover, each corner can position itself as top or bottom, left or right corner.

Processors in the underlying mesh may learn which of their corresponding edges belong to a row and which to a column in the following way. Corner processors may send their row/column decisions up their trees, and the corresponding four roots may return these data downward, informing border processors (i.e. nodes on the boundary of the underlying mesh connected computer) about their row and column edges (which is learned after exchanging root data among neighboring nodes). Border processors may repeat the process to inform internal nodes of the mesh about their row and column edges.

The computation of row and column numbers for each leaf node is not so obvious as the previous steps. The same procedure is performed in every row and every column simultaneously, and thus it suffices to consider one particular (say) row and find the column numbers of each processor in that
row. In order to understand the algorithm, we need to describe a relation of the binary trees over the row and the final binary addresses of nodes in the row. Let these addresses be given as numbers from 0 to $m - 1$ in the binary number representation. These binary addresses can be obtained as follows. Let left (right) edges be edges leading from a parent to the left (right, respectively) child. Mark each left edge in the binary tree with a 0 and each right edge by a 1. The binary address of a node is obtained by following the path from the root to that node, and taking the edge markings. Each of internal nodes in the row has two neighbors. One of the neighbors (the one which shares common parent) differs in the binary address in the last digit only while the other neighbors has certain number of the same digits followed by certain number of different digits. More precisely, two neighbors that do not share a common parent have their addresses as follows: $a_1, \ldots, a_k, 0, 1, 1, \ldots, 1$ and $a_1, \ldots, a_k, 1, 0, 0, \ldots, 0$ for some $k$ and some choice of $a_1, \ldots, a_k$. The paths from their lowest common ancestor in the tree to these nodes consist of one left (right) edge followed by certain number of right (left, respectively) edges only.

We shall now show how to use this property to find these binary addresses. First, each pair of two neighbors choose their common datum, which will be sent up the tree (for example, the average of their two data). Parents of leaf nodes receive two data each, which are obviously different, and forward both of them to their parents. Their parents (i.e. nodes at level 2, if leaves are at level 0) receive four data each. However, two of these data are the same, because the node is the lowest common ancestor of two nodes whose binary addresses end by 01 and 10, respectively. The other two data are forwarded to their parents (i.e. nodes at level 3) and each of the two data is returned to their children (each to the same child who sent the corresponding datum). These children (i.e. nodes at level 1) then return the datum (which came back) to the child they got it from, and are able to identify that the edge they are using to return has the same mark (left or right edge) as the edge that connects the node to its parent. This is the key observation in the algorithm: nodes are able to identify which of their two edges toward children has the same mark (i.e. orientation) as the edge leading toward their parents. The same process continues toward the root of the tree, at which point all nodes are able to mark their children if the edge toward their parents is marked by chance. However, the root node knows which of its children is left and which one is right, because corner nodes have made their decision (and that
decision may arrive during our process). In the next phase, the message passing goes from the root toward the leaves. Internal nodes receive from their parents the message identifying whether they are left or right children. According to the received message and the former identification of the child who is marked in the same way, they are, in turn, able to identify their left and right children. they can also pass the partial address of the node, collected along the path from the root to that node. In the end, leaves are able to identify their column addresses. The process requires $\Omega(\log m)$ time, whence the following theorem.

**Theorem 2.2.** Under the assumptions E1, L1, and N1, the task of initializing an $n$-processor mesh of trees can be performed in $\Theta(\log n)$ time.

If the nodes at each level of a complete binary tree are connected to form a linear array, a one-dimensional pyramid is obtained. More interesting is a two-dimensional pyramid, consisting of $n = (4^k - 1)/3$ processors located on $k + 1$ levels. All processors at the same level are connected to form a mesh. There are $4^i$ processors at level $i$, $0 \leq i \leq k$, arranged in a mesh-connected computer with $2^i$ rows and columns. A processor at level $i$, in addition to being connected to its four neighbors at the same level, also has connections to four children at level $i + 1$ and one parent at level $i - 1$, provided they exist. The only processor at level 0 is called the *apex*. The mesh at level $k$ is called the *base*.

The initialization process for pyramids starts from the four corners of the base mesh – these are the only processors with exactly three neighbors. Among their neighbors, their parents are the only nodes with seven edges. Such “climbing up” the pyramid continues until the apex is identified. Observe that in the very last step, each of the four corners at level 1 has, in addition to the apex two other neighbors also with seven edges; however, they already excluded themselves from being the apex. Once identified, the apex is able to decide on the mesh directions, taking into account the configuration of its four children, and on the number of nodes in the network. The addressing may easily continue downward, since each child of a node that just identified its address may decide on its level and row and column indices within the level on the basis of information from its parent and its (up to) four neighbors within the mesh at its own level.

**Theorem 2.3.** Under the assumptions E1, L1, and N1, the task of initializing a pyramid of $n$ processors can be performed in $\Theta(\log n)$ time.
3 Initializing Linear Arrays and Rings

The two end-processors of a linear array are initially leaders and begin sending a message towards the middle processors. It is clear that the two messages will meet at one processor (in case $n$ is odd) or at two processors (in case $n$ is even). In the latter case the two middle processors will choose a leader among themselves, and the new leader will decide about the left and right directions in the array. The total number of processors may easily be learned by the middle processors by counting the number of steps needed for the message to arrive from the end-processors. The elected leader then propagates the message back toward two end-processors, informing all the processors about their identity.

The simple initialization algorithm that we just described runs in $O(n)$ time under the most general assumptions E1, L1, and N1. The algorithm is asymptotically optimal since no processor inside the array may derive any conclusion about its position unless a message from an end-processor arrives. This observation yields an $\Omega(n)$ lower bound for the initialization problem on the linear array of processors. To summarize, we state the following result.

Theorem 3.1. Under the assumptions E1, L1, and N1 the task of initializing a linear array of $n$ processors can be performed in $\Theta(n)$ time. □

The initialization algorithm of a ring of processors will start by electing a leader: the same assumptions E1, L1 and N1 are taken. Each processor chooses one direction at random and sends its datum (which is unique for all processors) to its neighbor in the chosen direction. Any processor receiving one or two data from their neighbors will compare the incoming data with its own datum, and decide whether it remains a candidate for the leader. Each processor forwards any datum received from one of its neighbors on the opposite link to the other neighbor. The circulation stops when processors receive back their own datum, at which point they also learn the number of processors $n$ in the rink, corresponding to the number of data items seen. Exactly one processor will have seen data no smaller than its own – this processor declares itself the leader. The leader election algorithm on a ring is a variation of well known algorithm (cf. [1]).

Once the leader is elected, the initialization algorithm may proceed as described above for the linear array. A simple argument, similar to that presented for a linear array shows that the initialization algorithm is optimal. Thus, we state the following result.
\textbf{Theorem 3.2.} Under the assumptions E1, L1, and N1 the task of initializing a ring of $n$ processors can be performed in $\Theta(n)$ time. □

4 Direct Interconnection Networks

A direct interconnection networks consist of a set of processors, each having a local memory, and a set of bidirectional links that serve for the exchange of data between processors. They can be conveniently represented by undirected graphs, with processors as nodes and links as edges. In the previous section we have considered two of the simplest direct interconnection networks: linear arrays of processors and rings. We shall now generalize the ideas presented there and apply them to some of the other direct interconnection networks.

A general initialization algorithm consists of four main steps:

Step 1. Identify pairs of opposite edges in each node for each dimension. For example, on a 2-dimensional mesh or torus, two edges leading to the left and right neighbors are opposite, and also two edges towards the top and bottom neighbors,

Step 2. Find the number of nodes in the network,

Step 3. Apply a leader election algorithm (for instance, the one discussed in Lemma 4.1 below).

Step 4. Propagate node addresses from the leader.

The implementation of these steps depends on the particular network and on the underlying assumptions, especially on whether or not the edges are labeled. Additional steps may be needed for some models, while for other models some of these steps are trivial (for example, Step 1 on a ring) or meaningless (for example, Step 1 on a hypercube).

We shall describe a general algorithm for detecting a leader in any network. Let $deg$ be the network degree, that is the largest number of neighbors of each node. Let, further, $diam$ denote the network diameter, that is the maximum distance between any two nodes. In this context, the time complexity of the algorithm will be $O(deg \times diam)$. 

11
Initially, each node stores its own datum, distinct for all processors as discussed in Section 1 as the current maximum in the network. Repeat \(diam\) times the following step (we assume condition N2; for most networks the knowledge of the number of nodes and diameter are equivalent). In parallel, each node sends its current maximum to all its neighbors (it will take \(O(deg)\) time on one-port and \(O(1)\) on all-port model). Then each processor reads, one by one, the maxima received from its neighbors and compares them to its current maximum, updating it in the process (this will take \(O(deg)\) time). At the end, the global maxima will reach all nodes, and the unique node which was the source of that maximum will declare itself the leader. To summarize, we state the following general result.

**Lemma 4.1.** Under the assumptions L1, E1, and N2, the leader in a direct interconnection network can be found in \(O(deg \cdot diam)\) time. \(\square\)

If the network is asymmetric, then the leader election phase may be simplified by applying it to border processors only. Of course, this holds for models where all border processors are defining a connected graph. Furthermore, if edges are labeled (E2) then the leader election may become straightforward (say, the topmost and leftmost processor on a mesh).

We shall now define some direct interconnection networks that will be considered.

A \(k\)-dimensional mesh has \(n = r^k\) nodes corresponding to integer strings of the form \(c_1c_2\ldots c_k\), where \(0 \leq c_j \leq r - 1\) for \(1 \leq j \leq k\). Two nodes are linked by an edge if and only if they differ in exactly one coordinate and the absolute value of the difference is 1. The coordinate in which they differ is assigned as the link number to the corresponding edge. For \(k = 1\) this defines a linear array of processors, while for \(k = 2\) one obtains a mesh connected computer.

An \(k\)-dimensional torus has \(n = r^k\) nodes such that node \(c_1c_2\ldots c_k\), \(0 \leq c_i < r\), \(1 \leq i \leq k\), is connected to nodes \(c_1c_2\ldots c_{i-1}c_i \pm 1 \mod r\)\(c_{i+1}\ldots c_k\). The coordinate \(i\) in which the two node differ is the link number of the corresponding edge. For \(k = 1\) (and \(r > 2\)) one obtains a ring, while the case \(k = 2\) (and \(r > 2\)) defines a torus. The \(k\)-dimensional torus is obtained from the \(k\)-dimensional mesh by adding wrap-around edges between the first and the last processor in each dimension, so that the resulting network becomes edge and vertex symmetric. For instance, a two-dimensional torus (introduced in [11]) is obtained from a two-dimensional mesh connected computer by adding wrap-around edges in each row and each column.
$k$-dimensional hypercubes are obtained when $r = 2$ in the definition of a $k$-dimensional torus. Alternatively, a hypercube with $n = 2^k$ nodes can be defined as a network in which each node may be assigned an address with $k$ bits (each bit being 0 or 1) such that two nodes are connected by a link if and only if their addresses differ in exactly one bit. If the bit in which two neighbors differ is the $j$-th bit, ($1 \leq j \leq k$), then that link is referred to be link $j$, or link in direction $j$. Since the number of neighbors $k$ is known by each processor, the number of nodes $n$ in the network is known by processors in advance.

5 Initializing Meshes and Tori

We assume that edge directions are not known a priori (E1). Moreover, we want to achieve all desirable properties N1, E1, and L1. An initialization algorithm for meshes and tori may be obtained by an algorithm which consists of the four steps discussed in the previous section.

For simplicity, consider first two-dimensional meshes and tori in Step 1. We will show afterwards that the same procedure actually works correctly for $k$-dimensional meshes and tori. Each processor of a mesh or torus may identify two pairs of its opposite edges by the following algorithm. In parallel, each node sends its distinct datum to all its neighbors. In the next step, each node forwards all received data to all of its neighbors (except possibly to the ones the data came from). These two steps take a constant number of communications between nodes, because of the network’s constant degree. Now, each processor compares all data received and counts how many times each datum was received. Of particular interest are counters of value 2. If the origin of a datum has coordinates $(0, 0)$ then there will be up to four nodes which received that datum twice: nodes $(1, 1)$, $(1, -1)$, $(-1, 1)$ and $(-1, -1)$. For instance, node $(1, 1)$ received one copy of datum from $(0, 0)$ via node $(0, 1)$ and the other copy via node $(1, 0)$. Each node returns the data received in exactly two copies (together with its own datum) back toward the processor which owns the datum, following the previous two steps in the reverse order. Processors then verify what was received on each of its edges. Two opposite edges do not return counters 2 from the same processor. For instance, if one edge of node $(0, 0)$ returned counter from nodes $(1, 1)$ and $(1, -1)$ then its opposite edge is the one that returned the counter from nodes
\((-1,1)\) and \((-1,-1)\). In this way two pairs of opposite edges are identified by each processor. The special cases involving corner or border processors on a mesh can be easily handled. The algorithm for identifying the opposite edges on two-dimensional meshes and tori has time complexity \(O(1)\).

The same algorithm actually works for \(k\)-dimensional meshes and tori. The nodes that received exactly two copies of a datum from node \((0,0,\ldots,0)\) are exactly the nodes with two coordinates equal to \(\pm 1\) and other coordinates equal to 0. Two opposite edges in dimension \(j, 1 \leq j \leq k\), are recognized as such if and only if they do not return data from any common processor; more precisely, one of the edges returns from processors having \(j\)-th coordinate equal to 1 (and one more coordinate \(\pm 1\), the others being 0) while the other returns from processors having \(j\)-th coordinate \(-1\) (and one more coordinate \(\pm 1\), the others being 0). The complexity analysis and the details of the algorithm are straightforward and are therefore omitted. The algorithms runs in \(O(k^2 \log k)\) time, which can be viewed as a constant since the models normally consider the dimension \(k\) fixed, while only the row size \(r\) varies. It is important to note that, in the case of meshes, the border processors may also use the same procedure and find pairs of opposite edges and edges that do not have the matching one.

Step 2 can be implemented as follows. On a mesh, border processors will send a message on all their edges that do not have the opposite one (i.e. last edges in a given direction). Each processor, upon receiving any message on one edge, will forward it on the opposite edge. If there is no opposite edge (i.e. if the message reached the opposite side of the mesh) then the size of the row (i.e. \(r\)) is discovered and can be returned to internal processors using reverse paths. The dimension \(k\) is obviously learned by internal processors by counting the number of their edges. On a torus, each processor chooses one of its edges at random and sends its datum on it. Each processor receiving several data will forward each of them on the corresponding opposite edge. The row size \(r\) is corresponds to the number of steps needed for the datum to return to its own processor.

Step 3, electing a leader, may now be performed by applying Lemma 4.1 directly, since the number of nodes in the network and, therefore, its diameter have just been learned in Step 2.

In Step 4, the leader of a mesh or torus decides arbitrarily on the link numbers of its edges (i.e. on dimensions \(1,2,\ldots,k\)) and on the orientations of each edge, i.e. identifies the left, right, top, and bottom directions on a two-
dimensional mesh/torus or left and right orientations along each direction on a $k$-dimensional mesh or torus. On a mesh, the leader may send a message to the left in each of $k$ directions, looking for a node which is leftmost in each direction. The message is forwarded by each intermediate node using their opposite edges until a border processor is found. The number of edges traversed to the left in each direction suffices for the leader to identify its address (the $k$ border processors which are reached may return the counters using reverse paths). On a torus, the leader may choose its own address, say $(0,0,\ldots,0)$.

We consider first two-dimensional meshes and tori and proceed to generalize the algorithm afterwards. The row address of the leader may be broadcast to all processors in that row, and similarly for its column address. These processors in the row and column containing the leader may learn their other coordinate as well, based on the distance from the leader. In the next step, all processors that learned their address will initiate broadcasting its other coordinate (the one not shared with the leader) to processors in their columns (for processors in the same row as the leader) or row (for processors in the same column as the leader). Each remaining processor in the mesh or torus will receive two messages, one identifying its row and the other identifying its column address.

To describe a general algorithm for $k$-dimensional meshes and tori (and also hypercubes below), let the leader address be $(a_1a_2\ldots a_k)$. The leader sends a message (containing its own address, link number and orientation of the edges) to all its neighbors, which is sufficient for them to identify their address. These nodes continue to transmit the message on their opposite edges to incoming one, until all processors which differ from the leader in exactly one coordinate discover their addresses. Next, in $r-1$ iterations (for $j=1,\ldots,r-1$), the following steps will be performed. Each processor that identified its address in the previous iteration (these are exactly processors having exactly $j$ coordinates different from the coordinates of the leader) will send its address (and address of the leader) on all their links. Note that the link numbers are not yet known. The message is forwarded from receiving nodes on their opposite edges, and these nodes learn their distance from the node that sent the message. After getting first two messages from any two neighbors, the processor may identify its address. Note that the addresses of these two neighbors differ in exactly two coordinates; each of them has one of the two coordinates the same as in the leader, which is exactly the
link number of the corresponding edge. The link numbers, once identified, are returned to all nodes along given direction. The presence of messages from other neighbors will identify the corresponding link numbers of the connecting edges.

We summarize by stating the following results.

**Theorem 5.1.** The task of initializing a 2-dimensional mesh or a torus of \( n \) processors under the assumptions E1, L1, and N1 can be performed in \( O(\sqrt{n}) \) time. \( \square \)

**Theorem 5.2.** The task of initializing a \( k \)-dimensional mesh or torus of \( n \) processors under the assumptions E1, L1, and N1 can be performed in \( O(k^2 n^{1/k}) \) time.

**Proof.** Steps 1–4 of the algorithm require \( O(k^2 \log k), O(r), O(k^2 r), \) and \( O(k^3 + k^2 r) \) time, respectively, where \( n = r^k \). \( \square \)

6 Initializing a Hypercube and a Butterfly

A hypercube initialization algorithm does not need Steps 1 and 2 from the previous section since there are only two nodes in each direction (i.e. no left or right orientation) and the number of nodes \( n = 2^k \) is clearly calculated by counting the number of incident edges \( k \) at each node. The leader processor in a hypercube may be determined in \( O(\log^2 n) \) time, following the general algorithm preceding Lemma 4.1. Broadcasting addresses from the leader follows the general algorithm discussed in the previous section. The leader declares itself to be processor 0 = 00...0, and decides on link numbers of each of its edges and sends a message to all its neighbors, informing them about their address. Next, in \( \log n \) iterations, the following steps will be performed. Each processor that just identified its address will send its address on all its links. Depending on the adopted assumptions, this will require \( O(\log n) \) time in the one-port model and \( O(1) \) time in the all-port model. All processors will check whether something arrived from each of their neighbors; this will require \( O(\log n) \) time, assuming that processor have no other means to detect whether they received a message from any of its neighbors but to simply check \( \log n \) memory locations that are designated for receiving messages from neighboring nodes, one by one. Therefore even when only two neighbors have sent a message, it still takes \( O(\log n) \) time to find and read them. After getting first two messages from any two neighbors, the processor may identify
its address as the bitwise OR of the addresses of the two neighbors. The link
numbers of the two edges leading to the two neighbors are also identified.
The presence of messages from other neighbors will identify the corresponding
link numbers of the connecting edges. The time complexity of the algorithm
is therefore \( O(\log^2 n) \), assuming one-port model and L1, N1 and E1.

**Theorem 6.1.** The task of initializing a hypercube of \( n \) processors under
the assumptions E1, L1, and N1 can be performed in \( O(\log^2 n) \) time. \( \Box \)

It is not clear whether the algorithm we just discussed is optimal, since
the only lower bound available at the time of this writing is \( \Omega(\log n) \). We
observe that the described algorithm becomes optimal \( O(\log n) \) time under
the following assumptions:

- the leader is known (L2),

- the bitwise OR can be computed in constant time, and

- the processors are able, in the presence of any arriving message, to
  immediately detect which neighbors sent the message and read these
  messages at once.

It is important to note that the initialization algorithm for hypercubes
can be made optimal if the edge labels are known in advance (E2), as follows.
The leader election process may be performed by an algorithm that finds the
maximum elements among data which are distributed one per processor. The
data for which the maximum is to be found are exactly the distinct values
that processors have, as discussed in the introduction. The well known algo-
rithm for finding the maximum element for some interconnection networks,
including torus and hypercube, has the property that processors do not need
to know their identity (i.e. address) in order to run the algorithm. It suffices
to know the edge orientations. These maximal element algorithms can be
described as follows.

For each direction, in parallel, shift all data in the given direction until
all processors in the given direction learn the maximum among the elements
being shifted.

This algorithm will simplify the initialization algorithm described above
for meshes and tori. We omit the details and turn our attention to hyper-
cubes. The maximal element algorithm on a hypercube can be described as
follows.
For $j = 1$ to $k$, in parallel, each processor reads the current maximum of its neighbor on link $j$, and chooses the larger between its current maximum and the received number from the neighbor as the new current maximum.

The leader assigns the address $0 = 00\ldots0$ to itself. Indices of the other processors can be determined by following broadcasting algorithm. The broadcasting problem for a given network is defined as the problem of sending the same message from a source node to all other processors in the network. In our case there is no single message that is to be received by all processors, but rather we use the path created by the broadcasting procedure, and, instead of the fixed message, processors send their newly discovered address to their neighbors. That address, together with knowledge of the direction of the edge joining two neighbors, suffices for the neighboring node to identify its own address. The broadcasting algorithm for hypercubes can be described in the following way.

For $j = 1$ to $k$, processors that learned their identity so far, send their address to the neighbor on link $j$; that neighbor changes the $j$-th bit in its address from 0 to 1 to establish its identity. All processor will learn their identity in $O(\log n)$ time.

We summarize the previous discussion by stating the following result.

**Theorem 6.2.** The tasks of initializing an $n$-processor hypercube under the assumptions E2, L1 and N1 can be performed in time $\Theta(\log n)$. □

A $k$-dimensional butterfly network (called also Banyan, baseline, bidelta, flip, omega etc. in literature) can be obtained from a hypercube by replacing each node with a linear array of $k + 1$ processors. The processors are conveniently denoted as $cl$, where $c = c_1c_2\ldots c_k$ and $0 \leq l \leq k$. Node $cl$ is connected to the following two (for $l = 0$ and $l = k + 1$) or four (for $0 < l < k + 1$) nodes: $c(l + 1), \oplus (c, l)(l + 1), c(l - 1)$ and $\oplus (c, l - 1)(l - 1)$, where $\oplus (c, l)$ denotes node $c$ with $l$-th bit changed (i.e. 0 to 1 and vice versa; more precisely, $\oplus (c_1c_2\ldots c_k l) = c_1c_2\ldots c_{l-1}c_{l+1}\ldots c_k + 1 \pmod{2}c_{l+1}\ldots c_k$. A butterfly consists therefore of $2^k(k + 1)$ processors which are organized into $k + 1$ rows and $2^k$ columns. Each node in row $l$ is connected to two neighbors in row $l + 1$ and two neighbors in row $l - 1$. The diameter of the network is $2k$; every two nodes can be connected via a processor in row $k$. A hypercube can be obtained by reducing each column to one processor and keeping all connections that existed before the reduction.

Note that the columns can be reordered and rows numbered in reverse order, to obtain another butterfly network. Because of this property, a leader
election process should determine the role of the first and last rows. Nodes that have exactly two neighbors will send their datum on both their edges. Each internal node chooses the larger among two incoming data and forwards it to two neighbors at the next level. There are two simultaneous data streams, from first to last and from last to first row. After $k$ steps, each node in row 1 will receive the largest datum from nodes in row $k + 1$ and vice versa. The process continues for another $k$ steps, after which one border row will be “defeated” by a larger datum from another border row. This phase decides which row is row 1 (and all nodes in that row learn that).

It is important to observe that nodes at rows $1, 2, \ldots, d$ for any $d$, $1 < d < k + 1$ define $2^{k-d-1}$ mutually disconnected $(d - 1)$-dimensional sub-butterflies, each with $2^{d-1}d$ nodes. The initialization algorithm proceeds as follows. Nodes of each $d$-dimensional sub-butterfly update the largest datum among the corresponding nodes at row 0 as the row number $d$ progresses. For rows $d = 1, 2, \ldots, k$, the $d$-th bit of nodes in rows $1, 2, \ldots, d$ is determined as follows. Processors at level $d$ send their current maximum data to both of their neighbors at level $d + 1$. Nodes in row $d + 1$ compare these current maxima and assign 0 to larger and 1 to smaller of these data as the $d$-th bit of all nodes of $d - 1$-dimensional butterflies (these bits are propagated to nodes in rows $< d$). Also, nodes in row $d + 1$ choose the larger of incoming data for the next iteration of the algorithm.

**Theorem 6.3.** The tasks of initializing an $n$-processor butterfly under the assumptions E1, L1, and N1 can be performed in time $\Theta(\log n)$.

7  **Networks with Buses**

In this section, we consider linear arrays and rings enhanced by the addition of a bus that is accessible to all processors. Although all the processors can read the message broadcast on the bus, at any one time only one processor may broadcast on the bus. We also consider meshes augmented with multiple broadcasting capabilities [6]. In accord with other workers, we assume the transmission on buses to be ideal, with no broadcast delays [6, 9, 10, 13]. The assumption about the model is confirmed by experimental results [13].

Let us consider first the linear array with a global bus. The two end-processors play a special role, and they are able to use the bus for message transmission. If assumption E2 is taken then they can distinguish among
themselves as being the leftmost and rightmost processor, and the algorithm may easily give permission to one of them to use the bus. However, under assumption E1 they are not easily distinguishable because they are far apart to decide by themselves who is the leader. One possible resolution is to choose a number \( t \) and let both processor generate a random integer in the range \([1, t]\). Let \( t1 \) and \( t2 \) be generated, respectively. They decide to send a message on the bus at times \( t1 \) and \( t2 \), respectively. The first one to use the bus is the leader. The probability of using the bus simultaneously is \( 1/t \). The value of \( t \) may be assumed to be equal to the running time of the rest of algorithm, which in this case is \( \sqrt{n} \). Since we are developing deterministic, not randomized algorithms, we shall not accept this algorithm and will claim the assumption E2 for the algorithm, if leader is not chosen a priori (L1). Alternatively, if the master processor elects one of end processors as the leader (L2) then E1 suffices, because that leader can easily decide on the role of end processors.

Let us assume first that the number \( n \) of processors is known to all processors (assumption N2). We will describe an \( O(\sqrt{n}) \) initialization algorithm under this assumption, and will show afterwards how to modify the algorithm to run under assumption N1, that is if the number of processors is not known.

The first phase of the initialization algorithm proceeds as follows. Each processor uses its own distinguished datum to link itself to two other processors that are \( \sqrt{n} \) links apart. In parallel, each processor sends its datum to both of its neighbors. Then, in the next \( \sqrt{n} - 1 \) time units, each processor forwards the data received on the link opposite to the incoming one. As a result, after \( \sqrt{n} \) such communication steps, each processor has sent its datum to two other processors, at distance \( \sqrt{n} \) each. These data are virtual pointers between pairs of processors, called companions. These pairs (each processor participates in one or two pairs) may point to one another by matching companion’s address with the datum that is broadcast on the bus. The second phase starts from the left-end processor, which identifies itself as processor 1 and sends on the bus its own datum. Exactly one other processor, namely its only companion, will match the message on the bus with the datum of one of its own companions, and will identify itself as processor \( \sqrt{n} + 1 \). In step \( i, 1 \leq i \leq \sqrt{n} - 1 \), the processor that previously identified itself as processor \( i\sqrt{n} + 1 \) will send its datum on the bus. Its second companion (the one that did not yet learn its address), will match the message on the bus
with the datum of one of its companions, and will recognize itself as processor \((i + 1)\sqrt{n} + 1\). In the final phase, processors that were able to identify their address (local leaders) will send their address to both neighbors, and that address is broadcast using links between neighboring processors until it reaches the next local leader. After receiving addresses of two neighboring local leaders, and counting the steps at which it occurs, all processors are able to discover their address and also the left and right directions. Finally, local leaders may learn directions from their neighbors.

Now, how to determine \(n\) if it is not known to processors a priori? Follow the above procedure for \(k = 1, 2, 4, 8, 16, \ldots\) until the right-end processor receives its address i.e. after it is “caught” by \(\sqrt{k}\) pointer jumps. Clearly, this will happen when \(k = O(n)\). Once the right-end processor receives the address, it is sent on the bus to inform others about the number of processors \(n\) in the network. The total time is \(O(\sqrt{n})\).

Consider now the lower bound for the initialization problem. Suppose that an initialization algorithm exists which runs in \(t\) steps. In time \(t\), at most \(t\) processors have used the bus to send a message. Therefore there exist a gap of size at least \(n/t\) between two bus users. The processors in that gap may use only local links to identify their address. The messages which are broadcast on the bus, being the same for everyone, cannot distinguish between them. In time \(t\), at most \(2t\) processors may learn their address by getting some information from two neighboring bus users. Therefore the size of the gap is at most \(2t\), e.g. \(n/t \leq 2t\) which means \(t \in O(\sqrt{n})\). To summarize our findings we state the following result.

**Theorem 7.1.** Under the assumptions N1 and either L2, E1 or L1, E2, the task of initializing an \(n\)-processor linear array endowed with a global bus can be solved in \(\Theta(\sqrt{n})\) time. \(\square\)

Next, consider a ring-connected machine augmented with a global bus. If a leader is known (L2) then the previous algorithm can be applied. Otherwise, we are not aware of any algorithm that will work in faster than \(O(n)\) time because processors may not be able to decide on who will use the bus first in less than \(O(n)\) time.

**Theorem 7.2.** Under the assumptions L2, E1, N1, the task of initializing a ring endowed with a global bus can be solved in \(\Theta(\sqrt{n})\) time. \(\square\)

A *mesh with multiple broadcasting* (MMB, for short) is a mesh-connected computer of size \(\sqrt{n} \times \sqrt{n}\) enhanced with one bus in each row and each column \([6, 13]\). These buses operate in the same way as the bus described
for the linear array. The MMB is commercially available and has attracted a good deal of well-deserved attention in the literature [3, 6, 13].

An initialization algorithm for the MMB may be obtained as follows. Each processor may classify itself as either a corner, border or internal processor based on the number of its neighbors (2, 3, and 4, respectively). We shall exclude internal processors in the first phase. Apply the algorithm described for the linear array independently on each of four border rows and columns. Under the assumptions E2 or L2, the four corners will receive desirable addresses \((1,1), (1,n), (n,1)\) and \((n,n)\). At the end of this phase each corner and border processor will find the row and column indices to which they belong. In the second phase processors in the first row broadcast the column index to internal processors using column buses, followed by a similar step for row indices. Since the size of linear array is \(O(n^{1/4})\), the algorithm runs in \(O(n^{1/4})\) time.

**Theorem 7.3.** The task of initializing a mesh with multiple broadcasting of size \(\sqrt{n} \times \sqrt{n}\) can be performed in \(O(n^{1/4})\) time under the same assumptions as for the linear array with a global bus. 

A lower bound may be derived as follows. Suppose that an initialization algorithm exists which runs in \(t\) steps. In time \(t\), at most \(t\) processors have used any given bus to send a message. Therefore the total number of bus users is at most \(t\sqrt{n}\). Let us divide the mesh into one plus \(t\sqrt{n}\) square submeshes. By Dirichlet’s principle, at least one of these submeshes contains no bus user. The area of the submesh is \(\frac{n}{t\sqrt{n}} = \frac{\sqrt{n}}{t}\), and it has \(\frac{n^{1/4}}{\sqrt{t}}\) rows and columns. The processors in that submesh may use only local links to identify their address. The messages which are broadcast on the buses, being the same for everyone, cannot distinguish between them. The processor in the middle of the submesh needs \(O(t)\) time to receive its identity by local communications links only, therefore, up to a constant, \(\frac{n^{1/4}}{\sqrt{t}} \leq t\), which implies \(t \geq n^{1/6}\). Therefore the lower bound is \(\Omega(n^{1/6})\).

**Theorem 7.4.** Any algorithm that correctly initializes a mesh with multiple broadcasting of size \(\sqrt{n} \times \sqrt{n}\) must take \(\Omega(n^{1/6})\) time in the worst case.

8 Conclusions

It remains open to find initialization algorithms for models not mentioned in this paper. In addition, the initialization problems for MMB (assump-
tions E2, L1 or E1, L2) and hypercubes (assumption E1) do not match the corresponding lower bounds. Also, we are not able to describe any efficient algorithm for models with buses under both E1 and L1 assumptions. Another related problem is that of the recognition problem, in which processors are supposed to determine whether they belong to a specific kind of network (e.g. a hypercube). To the best of our knowledge, there are no parallel algorithms for the recognition problem in literature, although the problem also seems to be important. A sequential recognition algorithm for hypercubes is given in [2] while [14] described several distributed algorithms for the recognition problem (recognition of a tree, star, bipartite graph, ring and complete graph).

References


