Chapter 6

Parallel Implementations and Costs for Recognised Functions

A program in the HOPP model is written independently of the architecture on which it is executed, the implication being that only *one* program is written. However, the nature of the parallel implementation which is selected does depend, among other parameters, on the characteristics of the target architecture as discussed in Section 5.1.2. The hypercube, 2-D torus, s-ary tree and linear array have been studied as potential target topologies. The hypercube proved to be the most suitable topology for the set of recognised functions and the tree and the linear array proved to be inefficient. In this chapter, implementations for the recognised functions on the previously mentioned topologies are discussed and their costs are derived.

6.1 Data Communications on the Topologies

Communication costs constitute a significant part of the total cost of execution of parallel programs on distributed memory machines. For the purposes of the implementation scheme, five types of communication patterns have been identified. The communication algorithms for the hypercube topology are based on [SS89]. Some of the mesh algorithms are based on [YM89]. However, the algorithms in

[SS89] and [YM89] assume that processors can communicate with more than one neighbour simultaneously. As mentioned in Section 5.1.2, the HOPP model does not assume such a capability. The algorithms and the derivation of the associated costs reflect this assumption.

The following patterns of communications recur in programs.

1. Nearest Neighbour Communication

   This type of communication involves sending a message (or a data packet) to a neighbouring processor. Two processors are considered to be neighbours if they are connected by a direct communication link.

2. Broadcast Operation

   This operation involves moving the same data packet from one processor to all the other processors in the network.

3. Scatter Operation

   This operation involves moving different data packets (one to each processor), from one processor to all the other processors in the network.

4. Gather Operation

   This operation involves collecting the data packets distributed across the processors in the network, onto a single processor.

5. Total Exchange Operation

   This operation involves moving data packets from every processor in the network to every other processor.

As pointed out in [SS89], the gather operation is the dual of the scatter operation, i.e. the algorithm for the gather operation can be obtained by reversing the data paths in the scatter algorithm and vice-versa. Hence, the costs for both the operations are identical. In the following discussion, only the scatter operation is considered.

The total number of processors in each case is assumed to be p.

- Hypercube - The number of processors in a binary hypercube of dimension d is given by: $p = 2^d$

- Torus - The number of processors in a 2-dimensional torus, where,

  p_1 represents the number of processors in each row
  p_2 represents the number of processors in each column
  is given by: $p = p_1 p_2$, $p_1 = 2^{k_1}$, $p_2 = 2^{k_2}$, $p_1 \leq p_2$.

  The number of processors in each dimension is made a power of two in order to allow a direct comparison of the performance of the torus with that of the hypercube. The cost expressions would remain unchanged even with this restriction removed.

- Tree - The number of processors in a tree of arity s and depth d is given by: $p = \frac{s^{d+1}-1}{s-1}$

Figure 6.1 illustrates the processor-numbering schemes for examples of the four topologies.

The algorithms for performing communications on the hypercube topology are based on the corresponding ones in [SS89], are therefore not described here, with only the cost expressions being presented. The size of the data to be communicated is assumed to be n bytes and all the data is assumed to be initially resident on processor$_0$. This does not matter in the case of the hypercube; the algorithms would remain unchanged irrespective of the processor from which the data is broadcast, scattered or gathered. It must also be emphasised that the

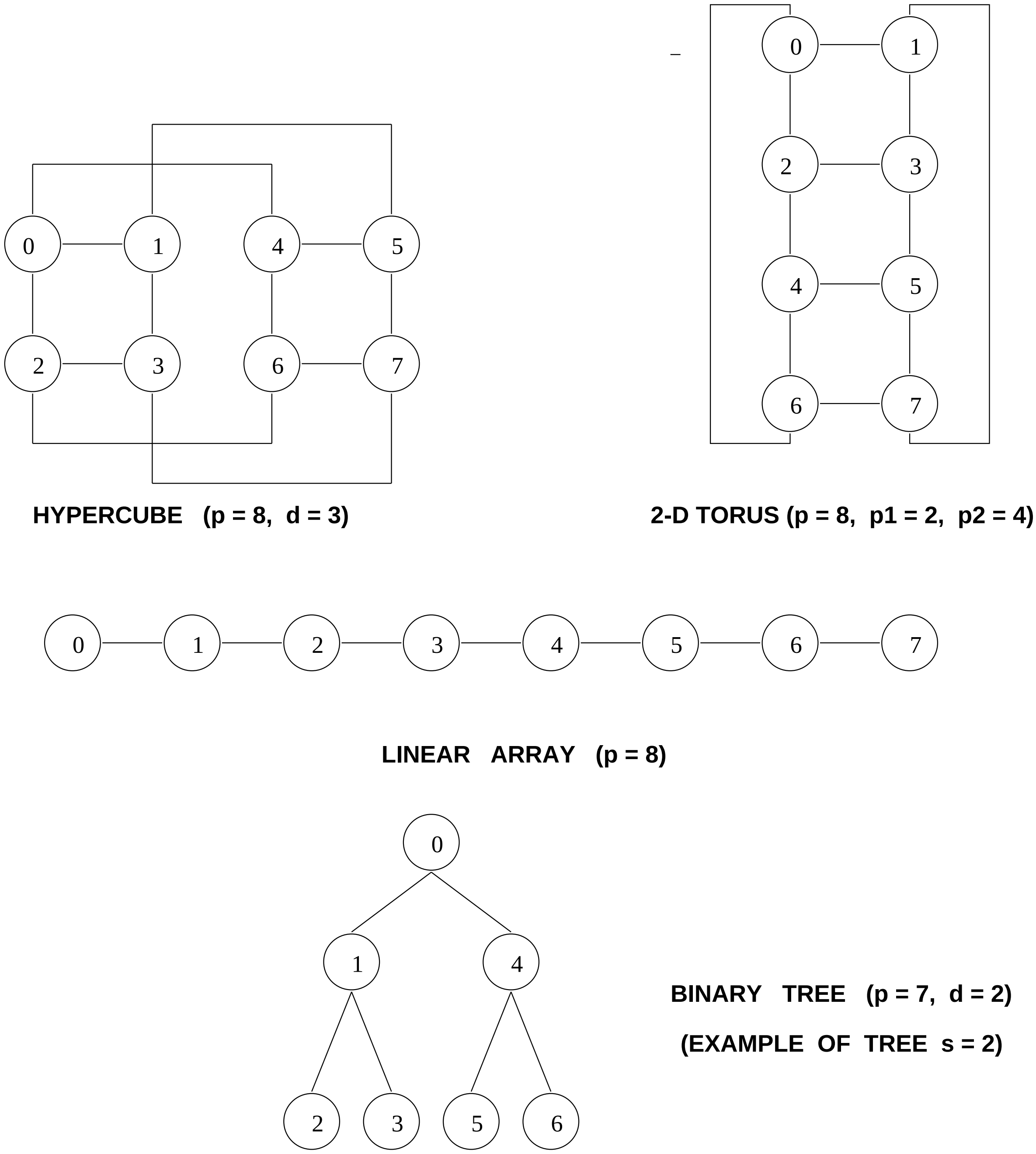

Figure 6.1: Processor-numbering Schemes for the Four Topologies

algorithms are not necessarily *optimal*, since the derivation of optimal communication algorithms is not central to this thesis. However, this will not affect the results of the thesis; a more cost-effective data distribution algorithm would probably result in a lower cost of implementation and could be incorporated in the scheme even at a later stage.

In the ensuing discussion, it is assumed that n, the number of elements in the input list, is divisible by the number of processors, p. If not, the cost computations would have to consider the expression $\lceil \frac{n}{p} \rceil$ and this would make it difficult to simplify cost expressions. If n is not divisible by p, then without loss of generality, it can be replaced by n', where $n' = (n+k)$, $0 < k < p$ and $(n+k) \bmod p = 0$. The costs are computed with the number of elements equal to n'. In all the cost expressions, it is assumed that the necessary substitution has already been made.

6.1.1 Nearest-neighbour Communication

The cost expression for this operation is identical on all the topologies. The operation itself is very simple - processor A sends a data packet to a neighbouring processor B which receives it. Using the linear model of communication, the cost for communicating a data packet of size n bytes to a nearest neighbour is given by:

$$C_{nn} = K_0 + \frac{1}{K_1} n \tag{6.1}$$

6.1.2 The Broadcast Operation

1. Hypercube - The cost of broadcasting a data packet of size n bytes on a hypercube of dimension d is given by:

$$C^h_{broad} = d(K_0 + \frac{1}{K_1} n). \tag{6.2}$$

2. Linear Array - The cost of communication on the linear array is more expensive because of its poor connectivity. In order to exploit parallelism in

the links, a data packet of size n bytes is split into r equal-sized packets and the communication of these packets are overlapped on different links. The overlap is possible only when $p > 3$, otherwise, the case is trivial and the data reaches the last processor in the linear array in two steps. For $(p > 3)$, the first data packet reaches the last processor in $(p-1)$ steps. Thereafter, since a *send* and a *receive* cannot be performed simultaneously in the assumed model, the last processor receives a data packet every alternate step. Since there are $(r-1)$ packets still to be received, a total of $2(r-1)$ steps will be required for the last processor to receive all the data. The other processors would have received all the data by this time. Since the amount of data transferred in each step is $\frac{n}{r}$, the cost of communication is given by:

$$C^{a}_{broad} = ((p-1) + 2(r-1))(K_0 + \frac{1}{K_1}\frac{n}{r}). \tag{6.3}$$

The expression simplifies to:

$$C^{a}_{broad} = (p + 2r - 3)(K_0 + \frac{1}{K_1}\frac{n}{r}). \tag{6.4}$$

The optimal value of r is given by: $r = \lceil\sqrt{\frac{(p-3)n}{2K_0K_1}}\rceil$.

3. 2-D Torus/Mesh - The data is first pipelined vertically and in the second stage of the broadcast, each row of processors pipelines the data horizontally in parallel. The wrap-around connections can be used to reduce the cost of broadcast in a 2-D torus. Instead of pipelining the data down the first column, processor$_0$, in the first step sends the data to the last processor in the column, using the wrap-around link (See Figure 6.1). In the second step, both the processors pipeline the data in opposite directions until all the processors in the first column receive the data. This reduces the length of the pipeline by half. A similar strategy is repeated along each of the rows in parallel in the next step. The total cost is then obtained by adding the

costs for each of the three stages.

$$C_{broad}^{m\_ideal} = (K_0 + \frac{1}{K_1}n) + (\frac{p_2}{2} - 1)(K_0 + \frac{1}{K_1}n) + (K_0 + \frac{1}{K_1}n) + (\frac{p_1}{2} - 1)(K_0 + \frac{1}{K_1}n)$$

The total cost of broadcast in a mesh is given by:

$$C_{broad}^{m\_ideal} = \frac{p_1 + p_2}{2}(K_0 + \frac{1}{K_1}n). \tag{6.5}$$

There is a slight non-uniformity in the 2-D torus considered for implementation. The example programs are to be implemented on a transputer-based machine. The transputers are configured as a 2-D torus for the purposes of implementation. Each transputer has only four links and each processor in a 2-D torus has at most four neighbours, so the logical neighbours on the torus can be made physical neighbours on the transputer network. However, an extra link on one processor is required to communicate with the operating system. This requirement causes a slight non-uniformity in the configured torus architecture and, as a result, processor$_0$ is physically connected to only three neighbours. This increases the cost of the broadcast operation since the data communication to one of processor$_0$'s neighbours (processor$_{p_1-1}$, in this case), cannot be performed in one step. This implies that the wrap-around connection cannot be used in the first row, to reduce the length of the pipeline by half. The cost expression is therefore modified to the following.

$$C_{broad}^{m} = (p_1 + \frac{p_2}{2} - 1)(K_0 + \frac{1}{K_1}n) \tag{6.6}$$

4. s-ary Tree - The broadcast operation in the case of the tree is straightforward. The data to be broadcast is initially at the root of the tree. The root sends the data to one child at a time, starting from the left to the right. Each child node receives the data from its parent and communicates

it to its children, starting with the leftmost child. The communications performed by the nodes in each level are overlapped, resulting in s such communications. For a tree of depth d, the cost is given by:

$$C^{t}_{broad} = sd(K_0 + \frac{1}{K_1}n). \tag{6.7}$$

6.1.3 The Scatter Operation

The data is divided into p equal parts which are numbered 0,1,2,..., $(p-1)$. After the scatter operation, each processor has $\frac{n}{p}$ bytes of the data. It is important that each processor receives the correct portion of the data. In particular, $processor_i$ should receive the i-th part of the data. (For processor numbering patterns, see Figure 6.1). This is because the algorithms for the parallel implementations of the recognised functions are based on a particular node numbering and assume the correct placement of data. Since the arguments of none of the recognised functions are assumed to be commutative, incorrect results may be obtained if the data is not scattered in the required order.

1. Hypercube - The cost of scattering data on a hypercube of dimension d is given by:

$$C^{h}_{scatter} = dK_0 + \frac{1}{K_1}\frac{n}{p}(p-1). \tag{6.8}$$

2. Linear Array - The data is just sent down the array and each processor retains the first $\frac{n}{p}$ bytes of the data, sending the rest to its neighbour. There is no parallelism exploited, but it is unlikely to improve the performance if communication is overlapped on the different links. The communication involves $(p-1)$ steps, and in each step, the amount of data reduces by $\frac{n}{p}$ bytes. The total cost is therefore given by:

$$C^{a}_{scatter} = \sum_{i=1}^{p-1}(K_0 + \frac{1}{K_1}(n - i\frac{n}{p})).$$

The cost expression simplifies to:

$$C^{a}_{scatter} = (p-1)(K_0 + \frac{1}{K_1}\frac{n}{2}). \quad (6.9)$$

3. 2-D Torus/Mesh - In the first step, the data is divided into two equal (or nearly equal) parts, with one half being sent from $processor_0$ to the last processor in the first column, using the wrap-around link. Each processor has $\frac{n}{2}$ bytes of the data after the first step. Each retains the appropriate amount of data ($\frac{n}{p_1}$ bytes), and in the second step, both the processors pipeline the data in opposite directions. At the end of step two, each processor in the first column has the data necessary ($\frac{n}{p_1}$ bytes) for its respective row. In the third step, all the processors in the first column repeat a similar procedure by sending data along the rows in parallel. The amount of data transferred along the links decreases with each transfer, by ($\frac{n}{p_1}$) bytes in step two, and by ($\frac{n}{p}$) bytes in step three. At the end of step three, each processor has the required portion of the data. Using the results for a linear array for the communications in steps two and three, the cost is given by:

$$\begin{aligned} C^{m\_ideal}_{scatter} &= (K_0 + \frac{1}{K_1}\frac{n}{2}) + (\frac{p_2}{2} - 1)(K_0 + \frac{1}{K_1}\frac{n}{4}) + \\ & (K_0 + \frac{1}{K_1}\frac{n}{2p_2}) + (\frac{p_1}{2} - 1)(K_0 + \frac{1}{K_1}\frac{n}{4p_2}). \end{aligned}$$

On simplification, the cost is given by:

$$C^{m\_ideal}_{scatter} = K_0(\frac{p_1+p_2}{2}) + \frac{1}{K_1}\frac{n}{4}((\frac{p_2}{2}+1) + \frac{1}{p_2}(\frac{p_1}{2}+1)). \quad (6.10)$$

If the non-uniformity in the wrap-around connections is accounted for, then in step three, the entire length of the pipeline must be considered. So, the cost expression modifies to the following.

$$\begin{aligned} C^{m}_{scatter} &= (K_0 + \frac{1}{K_1}\frac{n}{2}) + (\frac{p_2}{2} - 1)(K_0 + \frac{1}{K_1}\frac{n}{4}) + \\ & (p_1 - 1)(K_0 + \frac{1}{K_1}\frac{n}{2p_2}) \end{aligned}$$

The final cost expression for the non-uniform mesh simplifies to:

$$C^{m}_{scatter} = K_0(p_1 + \frac{p_2}{2} - 1) + \frac{1}{K_1}(\frac{n}{4}(\frac{p_2}{2} + 1) + \frac{n}{2p_2}(p_1 - 1)). \tag{6.11}$$

4. Tree - A parent node communicates data to each of its s children in turn, starting with the leftmost child. Each child node retains the first $\frac{n}{p}$ bytes of the data, being the data that is meant to reach that processor. The rest of the data is divided into s parts and communicated to its children. This scheme ensures that the data reaches the correct processor, since the trees are numbered by pre-order traversal (refer to Figure 6.1).

   At level j in the tree, each processor has to communicate the data it received from its parent, to k processors below it, where

$$k = s\frac{s^{d-j} - 1}{s - 1}. \tag{6.12}$$

   Each processor must possess $\frac{n}{p}$ bytes of the data after the scatter operation. So the amount of data still to be communicated at level j in the tree is $k\frac{n}{p}$. This data is divided into s parts and sent to each of the children. Each such communication involves the transfer of $\frac{k}{s}\frac{n}{p}$ bytes of data. Since there are s such communications in each level and there are d levels in the tree, the total cost is given by:

$$C^{t}_{scatter} = \sum_{j=0}^{d-1} s(K_0 + \frac{1}{K_1}(\frac{k}{s}\frac{n}{p})). \tag{6.13}$$

   Substituting for k from Equation 6.12 and simplifying the summation yields:

$$C^{t}_{scatter} = sdK_0 + \frac{1}{K_1}\frac{n}{p}\frac{s}{s-1}(p - 1 - d). \tag{6.14}$$

6.1.4 Total-Exchange

It is assumed that all the processors exchange equal amounts of data, i.e. $\frac{n}{p}$ bytes.

1. Hypercube - The cost of the total-exchange operation on a hypercube of dimension d is given by:

$$C^h_{exchange} = 2dK_0 + 2\frac{1}{K_1}\frac{n}{p}(p-1). \tag{6.15}$$

2. Linear Array - The total exchange operation on the linear array is very expensive due to its poor connectivity. The procedure involves dividing the linear array into two sub-arrays. In the first step, the processors in the right sub-array move their data to the left, and the processors in the left sub-array move their data to the right, in parallel. Hence, data from the two sub-arrays is gathered in the first processor of the right sub-array and the last processor of the left sub-array, respectively. In the second step, these two processors exchange the data. In the third step, the data is passed back in opposite directions along both the sub-arrays.

   The first step is just a gather operation involving half the number of processors in the original linear array and half the original data. The second step will therefore involve an exchange of $\frac{n}{2}$ bytes of data. In the third step, each processor also appends its own data to the data it receives and sends it on to its neighbour. With each communication, the amount of data which is transferred progressively increases by an amount equal to $\frac{n}{p}$ bytes. The total cost is obtained by adding the costs for the three steps. The number of processors, p, is assumed to be even. The cost is given by:

$$\begin{aligned} C^a_{exchange} &= (\frac{p}{2}-1)(K_0 + \frac{1}{K_1}\frac{n}{p}) + 2(K_0 + \frac{1}{K_1}\frac{n}{2}) + \\ &\quad \sum_{i=1}^{\frac{p}{2}-1}(K_0 + \frac{1}{K_1}(\frac{n}{2} + i\frac{n}{p})). \end{aligned}$$

On simplification, the cost expression is given by:

$$C^{a}_{exchange} = p(K_0 + \frac{1}{K_1}\frac{n}{2}). \tag{6.16}$$

3. Mesh - The method adopted is based on the one described in [YM89]. Each column and row of the mesh forms a ring because of wrap-around. In the first step of the total exchange operation, all the nodes exchange data with their column neighbours, by moving the data round the vertical rings. In the second step, a similar procedure is repeated along the rings in the rows. The communication in the columns (rows) are performed in parallel. Since $\frac{n}{p}$ bytes of data are involved in each transfer, the cost is given by:

$$C^{m\_ideal}_{exchange} = (K_0 + \frac{1}{K_1}\frac{n}{p})(p_2 - 1) + (K_0 + \frac{1}{K_1}\frac{n}{p_1})(p_1 - 1). \tag{6.17}$$

which on simplification yields:

$$C^{m\_ideal}_{exchange} = K_0(p_1 + p_2 - 2) + \frac{1}{K_1}n(1 - \frac{1}{p}). \tag{6.18}$$

However, for the case of the non-uniform mesh, the wrap-around connection in the first row cannot be used. So, two traversals of the first row are required. Since the number of processors along the rows (p_1), may be less than the number of processors along the columns (p_2), it proves to be more cost-effective if the data is exchanged along the rows in the first step, and along the columns, in the second step. The cost is given by:

$$C^{m}_{exchange} = (K_0 + \frac{1}{K_1}\frac{n}{p})(2p_1 - 2) + (K_0 + \frac{1}{K_1}\frac{n}{p_2})(p_2 - 1).$$

On simplification,

$$Cost^{m}_{exchange} = K_0(2p_1 + p_2 - 3) + \frac{1}{K_1}\frac{n}{p}(p_1 + p - 2).$$

Equation 6.16 could be used in the first step to further reduce the cost.

4. Tree - The method adopted involves gathering the results to the root processor and then broadcasting the results to all the processors. Since the cost of the gather operation is equal to the cost of the scatter operation, the cost is given by (refer to Equations 6.14, 6.7):

$$C^t_{exchange} = C^t_{scatter} + C^t_{broad} \tag{6.19}$$

6.2 Algorithms and Costs for the Parallel Implementations of Recognised Functions

Although the functions in the extended set can be expressed in terms of one or more functions in the basic set, their parallel implementation will often be different from that of the constituent functions. This is because the definitions for these functions in terms of the basic set of functions is often quite contrived and the corresponding implementations may not be cost-effective. In computing the costs of the recognised functions, the following assumptions are made.

1. Initial data distribution costs are ignored. These costs, which comprise of the costs for operations such as *scatter* or *broadcast*, are accounted for when the costs are computed for the entire problem.

2. The number of processors in the parallel machine is represented by p.

3. The input list is represented by xs, where $\mid xs \mid = n$

4. The input list, xs, is assumed to be distributed across the p processors in the parallel machine. The following assumption is applicable to all the algorithms, unless otherwise stated.

$$\begin{aligned} xs &= xs_0 @ xs_1 @ \ldots @ xs_{p-1} \\ & xs_i \in \text{processor}_i \\ & \mid xs_i \mid = \tfrac{n}{p}; \qquad ; i = 0, \ldots, (p-1) \end{aligned}$$

5. T^m_{com} represents the cost of communicating m elements of the list to a neighbour.

$$T^m_{com} = K_0 + \frac{1}{K_1} ms \qquad (6.20)$$

where s is the size (in bytes) of each element of the list.

6. The sequential argument function of a recognised function is represented by f and its cost by C_f.

7. The analyser makes the following assumption for all the sequential functions in the program.

$\forall x_i \in xs; \qquad i \in 0, 1, \ldots, (n-2)$

Cost $(f \;\, x_i) \approx$ Cost $(f \;\, x_{i+1})$,

where, f is any sequential function in the program.

This is an important assumption implying the *regularity* of the application that is considered for cost analysis.

For each of the recognised functions, the sequential cost is first computed. The algorithm for the parallel evaluation of the function on each of the four topologies is then presented, along with their associated costs. For the purposes of presentation, the same ordering of recognised functions is followed as in Chapter 4.

6.2.1 map

The sequential cost of **map** is given by:

$$C^s_{map} = nC_f \qquad (6.21)$$

The parallel implementation for **map** on all the topologies is straightforward. All the processors perform the **map** operation in parallel on their respective local data. The result of the operation remains distributed across the network.

Result $= ys = ys_0 @ ys_1 @ \ldots @ ys_{n-1}$

where, $ys_i \in$ processor$_i$

$i = 0, 1, \ldots, (n-1)$

The algorithm for the parallel implementation of **map** on all the four processor topologies is as follows.

```
Algorithm PAR_MAP (all topologies)
BEGIN
        FOR every processor_i DO in PARALLEL
              map f xs_i
END.
```

There is no communication involved in the implementation. The cost expression for the parallel implementation of **map** on any p-processor topology is identical and is given by:

$$C_{map} = \frac{n}{p} C_f \tag{6.22}$$

6.2.2 fold

In a sequential implementation of **fold**, there are $(n-1)$ applications of the operator f. (For the purposes of implementation and cost computations, $f\ a\ x_0$ is not considered, since the result is x_0). The sequential cost of **s_fold** is:

$$C^s_{s\_fold} = (n-1) C_f \tag{6.23}$$

In the case of **g_fold**, the input list is at least a list of lists, i.e. $D \geq 2$. If the list is assumed to comprise of n sublists, each of size m, then two cases need to be considered for the cost of the sequential function f.

- The case where the cost of f is independent of the sizes of the input lists on which it operates. The cost function can then be represented by C_f, and the cost expression for **g_fold** is identical to that of **s_fold**.

$$C^{s_c}_{g\_fold} = (n-1)C_f \tag{6.24}$$

- The case where the cost of f depends on the sizes of the input lists on which it operates. The cost function in this case is represented by a function g, as follows:

$$C_g = g(l_1, l_2)$$

  where l_1 and l_2 represent the lengths of the first and second argument lists, respectively. It may be noted that since each sublist is assumed to comprise of m elements, assumption 7 is still valid. Therefore, $l_1 = l_2 = m$. The cost expression is given by:

$$C^{s_f}_{g\_fold} = \sum_{i=1}^{n-1} g(im, m) \tag{6.25}$$

For all the topologies, after a parallel implementation of **fold**, the result is left on processor$_0$ (or the root processor). The algorithms for the two versions of **fold**, viz. **s_fold** and **g_fold**, are the same. The costs are different, because in the case of **g_fold**, the size of the data communicated increases at each step and must be accounted for in the computation of T_{com}.

6.2.2.1 fold on the Linear Array

The algorithm is as described in PAR_FOLD (linear array). The algorithm assumes that the number of processors is a power of 2, otherwise Step 2 of the algorithm will need slight modification. The cost expressions are unchanged.

Algorithm <u>PAR_FOLD (linear array)</u>
BEGIN
1. FOR each processor$_i$ DO in PARALLEL
/* *evaluate partial results* */
result $\longleftarrow$ **fold** f a xs_i
2. $j \longleftarrow 1$
WHILE $j \leq \log p$
combine partial results between pairs of processors which are j hops away
/* *combination from right to left* */
$j \longleftarrow 2 * j$
END.

- **s_fold**

$$C^a_{s\_fold} = \underbrace{C_f(\frac{n}{p} - 1)}_{step\_1} + \underbrace{(T^1_{com}(1 + 2 + \ldots + 2^{\log p - 1}) + C_f \log p)}_{step\_2}$$

On simplification, the cost of **s_fold** is given by:

$$C^a_{s\_fold} = C_f(\frac{n}{p} - 1 + \log p) + T^1_{com}(p - 1). \tag{6.26}$$

- **g_fold** - After performing Step 1 of the algorithm, each processor is left with $\frac{n}{p}m$ elements. In each iteration of Step 2, the number of elements to be communicated increases.

For the case where the cost of f is a function of its input list sizes:

$$C^{a_f}_{g\_fold} \quad = \quad \underbrace{\sum_{i=1}^{\frac{n}{p}-1} g(im, m)}_{step\_1} + \underbrace{\sum_{i=0}^{\log p - 1} (2^i T^{2^i \frac{n}{p} m}_{com} + g(2^i \frac{n}{p} m, 2^i \frac{n}{p} m))}_{step\_2}.$$

$$\begin{aligned} C^{a_f}_{g\_fold} \quad = \quad & \sum_{i=1}^{\frac{n}{p}-1} g(im, m) + \sum_{i=0}^{\log p - 1} g(2^i \frac{n}{p} m, 2^i \frac{n}{p} m) + \\ & \sum_{i=0}^{\log p - 1} 2^i T^{2^i \frac{n}{p} m}_{com}. \end{aligned} \tag{6.27}$$

For the case where the cost of f is independent of the input sizes, the cost expression simplifies to the following:

$$C^{a_c}_{g\_fold} = \underbrace{C_f(\frac{n}{p} - 1)}_{step\_1} + \underbrace{\sum_{i=0}^{\log p - 1} 2^i T_{com}^{2^i \frac{n}{p} m} + C_f \log p}_{step\_2} .$$

Grouping together the computation and communication terms, the cost is:

$$C^{a_c}_{g\_fold} = C_f(\frac{n}{p} - 1 + \log p) + \sum_{i=0}^{\log p - 1} 2^i T_{com}^{2^i \frac{n}{p} m} . \qquad (6.28)$$

6.2.2.2 fold on the Hypercube

The **fold** operation on a hypercube of dimension d involves the following two steps.

1. $(\frac{n}{p} - 1)$ local applications of the argument function on each processor, in parallel.

2. d nearest neighbour communications of partial results to the root processor, and d applications of the argument function to partial results from pairs of processors. (The root processor is the smallest-numbered processor in the cube or subcube implementing the **fold**).

In the following algorithm, p_{no} represents the processor number, and p^j_{no} represents the number of the neighbouring processor in dimension j of the hypercube; $0 \le j < d$, $0 \le p_{no}, p^j_{no} < p$.

Algorithm <u>PAR_FOLD (Hypercube)</u>
BEGIN
1. FOR every processor$_i$ DO in PARALLEL
   result $\longleftarrow$ **fold** f a xs_i
/ combine partial results on all processors */*
2. FOR every processor$_i$ DO
   FOR $j \longleftarrow 0$ TO $d-1$ DO
      IF $p^j_{no} < p_{no}$ THEN
         send **result** to processor p^j_{no}
         EXIT
      ELSE
         receive **data** from p^j_{no}
         result $\longleftarrow$ f **result data**
END.

- **s_fold** - The cost of **s_fold** is given by:

$$C^h_{s\_fold} = \underbrace{C_f(\frac{n}{p} - 1)}_{step\_1} + \underbrace{d(T^1_{com} + C_f)}_{step\_2}$$

which, on rearrangement gives:

$$C^h_{s\_fold} = C_f(\frac{n}{p} + d - 1) + T^1_{com}d. \tag{6.29}$$

- **g_fold** - If the cost of the sequential argument, f, is a function of input sizes, the cost is given by:

$$C^{h\text{-}f}_{g\_fold} = \underbrace{\sum_{i=1}^{\frac{n}{p}-1} g(im, m)}_{step\_1} + \underbrace{\sum_{i=0}^{d-1} T^{2^i \frac{n}{p} m}_{com} + \sum_{i=0}^{d-1} g(2^i \frac{n}{p} m, 2^i \frac{n}{p} m)}_{step\_2}.$$

On rearrangement, the cost expression is given by:

$$C^{h\text{-}f}_{g\_fold} = \sum_{i=1}^{\frac{n}{p}-1} g(im, m) + \sum_{i=0}^{d-1} (T^{2^i \frac{n}{p} m}_{com} + g(2^i \frac{n}{p} m, 2^i \frac{n}{p} m)). \tag{6.30}$$

If the cost of the argument function is a constant given by C_f,

$$C^{h\_c}_{g\_fold} = C_f(\frac{n}{p} + d - 1) + \sum_{i=0}^{d-1} T_{com}^{2^i \frac{n}{p} m}. \tag{6.31}$$

6.2.2.3 fold on the 2-D Torus/Mesh

The algorithm is based on an ideal torus where each processor has four physical neighbours. It exploits the wrap-around links in order to reduce the computation cost by half. The right and left halves of each row are treated as linear arrays and an algorithm similar to PAR_FOLD(linear array) is executed in each half. The partial results in each row are combined using the wrap-around link, so that the first column processors now hold the new set of partial results. A similar procedure is repeated along the first column processors, to obtain the final result on processor$_0$.

- **s_fold** -

$$\begin{aligned} C^{m\_ideal}_{s\_fold} &= \underbrace{C_f(\frac{n}{p} - 1)}_{step\_1} + \underbrace{T^1_{com}(\frac{p_1}{2} - 1) + C_f \log \frac{p_1}{2}}_{step\_2} + \underbrace{(T^1_{com} + C_f)}_{step\_3} \\ &\quad + \underbrace{T^1_{com}(\frac{p_2}{2} - 1) + C_f \log \frac{p_2}{2}}_{step\_4a} + \underbrace{T^1_{com} + C_f}_{step\_4b} \end{aligned}$$

Grouping like terms together, the cost is given by:

$$C^{m\_ideal}_{s\_fold} = C_f(\frac{n}{p} + 1 + \log \frac{p_1}{2} + \log \frac{p_2}{2}) + T^1_{com}(\frac{p_1 + p_2}{2}). \tag{6.32}$$

```
Algorithm PAR_FOLD (ideal-mesh)
BEGIN
     1. FOR every processor_i DO in PARALLEL
             result <-- fold f a xs_i
     2. j <-- 1
        WHILE j ≤ log p1/2
             Right (Left) half processors DO
                  combine partial results between pairs of
                  processors j hops away
                  /* from left (right) to right (left) */
             j <-- 2 * j
     3. Last and first-column processors DO
             IF last-column processor THEN
                  send result to first processor in row
             ELSE
                  receive data from last processor in row
                  result <-- f result data
     4. IF first-column processor THEN
        a. j <-- 1
           WHILE j ≤ log p2/2
                Bottom (Top) half processors DO
                     combine partial results between pairs of
                     processors j hops away
                /* from top (bottom) to bottom (top) */
                j <-- 2 * j
        b. IF last processor THEN
                send result to processor_0
           processor_0 DO
                receive data from last processor
                result <-- f result data
END.
```

- **g_fold** -

$$
\begin{aligned}
C_{g\_fold}^{mf\_ideal} = {} & \underbrace{\sum_{i=1}^{\frac{n}{p}-1} g(im,m) +}_{step\_1} \underbrace{\sum_{i=0}^{\log\frac{p_1}{2}-1} (2^i T_{com}^{2^i fracnpm} + g(2^i \frac{n}{p}m, 2^i \frac{n}{p}m)) +}_{step\_2} \\
& \underbrace{T_{com}^{\frac{p_1}{2}\frac{n}{p}m} + g(\frac{p_1}{2}\frac{n}{p}m, \frac{p_1}{2}\frac{n}{p}m) +}_{step\_3} \\
& \underbrace{\sum_{i=0}^{\frac{p_2}{2}-1} (2^i T_{com}^{2^i \frac{n}{p_2}m} + g(2^i \frac{n}{p_2}m, 2^i \frac{n}{p_2}m)) +}_{step\_4a} \\
& \underbrace{T_{com}^{\frac{n}{2}m} + g(\frac{n}{2}m, \frac{n}{2}m)}_{step\_4b} . \qquad (6.33)
\end{aligned}
$$

$$
\begin{aligned}
C_{g\_fold}^{mc\_ideal} = {} & C_f(\frac{n}{p} + 1 + \log\frac{p_1}{2} + \log\frac{p_2}{2}) + \sum_{i=0}^{\log\frac{p_1}{2}-1} (2^i T_{com}^{2^i \frac{n}{p}m}) + T_{com}^{\frac{p_1}{2}\frac{n}{p}m} + \\
& \sum_{i=0}^{\log\frac{p_2}{2}-1} (2^i T_{com}^{2^i \frac{n}{p_2}m}) + T_{com}^{\frac{n}{2}m}. \qquad (6.34)
\end{aligned}
$$

If the mesh does not have a wrap-around connection on all the rows, then all the links will have to be traversed in the horizontal direction. However, in the vertical direction, the wrap-around link can still be used. Steps 2 and 3 of the algorithm would require minor modification. Then, it is easily shown that the costs are modified as follows:

$$
C_{s\_fold}^{m} = C_f(\frac{n}{p} + \log p_1 + \log\frac{p_2}{2}) + T_{com}^{1}(p_1 + \frac{p_2}{2} - 1). \qquad (6.35)
$$

$$
\begin{aligned}
C_{g\_fold}^{m\_f} = {} & \sum_{i=1}^{\frac{n}{p}-1} g(im,m) + \sum_{i=0}^{\log p_1 - 1} (2^i T_{com}^{2^i \frac{n}{p}m} + g(2^i \frac{n}{p}m, 2^i \frac{n}{p}m)) + \\
& \sum_{i=0}^{\log\frac{p_2}{2}-1} (2^i T_{com}^{2^i \frac{n}{p_2}m} + g(2^i \frac{n}{p_2}m, 2^i \frac{n}{p_2}m)) + \\
& T_{com}^{\frac{n}{2}m} + g(\frac{n}{2}m, \frac{n}{2}m). \qquad (6.36)
\end{aligned}
$$

$$C^{m\_c}_{g\_fold} = C_f(\frac{n}{p} + \log p_1 + \log \frac{p_2}{2}) + \sum_{i=0}^{\log p_1 - 1} 2^i T_{com}^{2^i \frac{n}{p} m} + \sum_{i=0}^{\log \frac{p_2}{2} - 1} 2^i T_{com}^{2^i \frac{n}{p_2} m} + T_{com}^{\frac{n}{2} m}. \quad (6.37)$$

6.2.2.4 fold on the Tree

Algorithm PAR_FOLD (s-ary tree)
BEGIN
  1. FOR every processor$_i$ DO in PARALLEL
        result $\longleftarrow$ **fold** f a xs_i
  2. FOR every leaf processor DO
        send **result** to parent
     FOR every non-leaf processor DO
        FOR $i \longleftarrow 1$ to s DO
            receive **data** from child$_i$
            result $\longleftarrow$ f **result data**
            IF (not root processor) THEN
               send **result** to parent
END.

For a tree of depth d, the partial results take d steps to reach the root processor. At each level, there are s communications and s applications of the function f in parallel.

- **s_fold** - The cost expression is given by:

$$C^t_{s\_fold} = C_f(\frac{n}{p} - 1 + ds) + ds(T^1_{com}). \quad (6.38)$$

- **g_fold** - The cost expression is given by:

$$C^{t\_f}_{g\_fold} = \sum_{i=1}^{\frac{n}{p}-1} g(im, m) + s \sum_{j=0}^{d-1} T_{com}^{\frac{s^{d-j}-1}{s-1} \frac{n}{p} m} + \sum_{j=0}^{d-1} \sum_{i=0}^{s-1} g(\frac{n}{p} m + i \frac{s^{d-j}-1}{s-1} \frac{n}{p} m, \frac{s^{d-j}-1}{s-1} \frac{n}{p} m) \quad (6.39)$$

If the cost of f is independent of the size of its arguments, then

$$C^{t\_c}_{g\_fold} = C_f(\frac{n}{p} + ds - 1) + s\sum_{j=0}^{d-1} T_{com}^{\frac{s^{d-j}-1}{s-1}\frac{n}{p}m}. \tag{6.40}$$

The first term in Equation 6.39 is obtained from Step 1 of the algorithm. At level j in the tree (the root is at level 0, the leaves are at level $d-1$), each processor would have to send data that has been gathered from $\frac{s^{d-j}-1}{s-1}$ processors. At the end of Step 1, each processor has $\frac{n}{p}m$ data items. Since there are s communications at each level, and there are d levels in the tree, the total communication cost is given by the second term in Equation 6.39. At each level in the tree, a processor makes s applications of the function f. Each processor makes the first application of f on its local data and the data gathered from the leftmost child. For the second application of f, the size of the first argument is the sum of the sizes of the arguments in the first call to f. The same reasoning is valid for all the s applications of f at any level in the tree. For a non-leaf node at level j, each child node would have gathered data from $\frac{s^{d-j}-1}{s-1}$ nodes. Since there are d levels in the tree, the total cost of the function applications is derived to be the third term in Equation 6.39.

6.2.3 scan

The sequential implementation of **scan** has $(n-1)$ applications of the operator f. The sequential cost of **scan** is, therefore, the same as the sequential cost of **fold**. As in the case of **fold**, $g(l_1, l_2)$ represents the cost of the argument of **g_scan**, when it is a function of the sizes of its input lists. Once again, $l_1 = l_2 = m$.

$$C^{s}_{s\_scan} = (n-1)C_f \tag{6.41}$$

$$C^{s_f}_{g\_scan} = \sum_{i=1}^{n} g(im, m) \tag{6.42}$$

$$C^{s_c}_{g\_scan} = (n-1)C_f \tag{6.43}$$

The algorithms for the parallel implementations of **s_scan** and **g_scan** are identical, although the cost expressions are different. The result of the parallel **scan** operation is distributed across the network as follows:

Result $= ys = ys_0 @ ys_1 @ \ldots @ ys_{n-1}$
where, $ys_i \in$ processor$_i$.

In the following algorithms, *last* is a function that returns the last element of a list.

6.2.3.1 scan on the Linear Array

The partial results reach the last processor in the array in (p-1) steps. The total cost is once again obtained by adding the costs for both the steps in order.

- **s_scan**

$$C^a_{s\_scan} = \underbrace{C_f(\frac{n}{p} - 1)}_{step\_1} + \underbrace{((p-1)(T^1_{com} + C_f) + C_f(\frac{n}{p} - 2))}_{step\_2}$$

On simplification, the total cost is given by:

$$C^a_{s\_scan} = C_f(2\frac{n}{p} + p - 4) + T^1_{com}(p-1). \tag{6.44}$$

Algorithm PAR_SCAN (linear array)
BEGIN
1. FOR every processor$_i$ DO in PARALLEL
  result $\longleftarrow$ **scan** f a xs_i
  last_result $\longleftarrow$ *last* **result**
2. processor$_0$ DO
  send **last_result** to processor$_1$
 FOR every processor except processor$_0$ DO
  receive **data** from left neighbour
  last_result $\longleftarrow$ f **data last_result**
  send **last_result** to right neighbour
  FOR $j \longleftarrow 1$ TO $(\frac{n}{p}-1)$ DO
   result$_j$ $\longleftarrow$ f **data result**$_j$
  result$_{\frac{n}{p}}$ $\longleftarrow$ **last_result**
END.

- **g_scan**

$$C^{af}_{g\_scan} = \underbrace{\sum_{i=1}^{\frac{n}{p}-1} g(m, im)}_{step\_1} + \underbrace{\sum_{i=1}^{p-1} (T_{com}^{i\frac{n}{p}m} + g(i\frac{n}{p}m, \frac{n}{p}m))}_{step\_2} + \underbrace{\sum_{i=1}^{\frac{n}{p}-2} g((p-1)\frac{n}{p}m, im)}_{step\_2} \tag{6.45}$$

For the input-independent case, the cost of **g_scan** is given by:

$$C^{ac}_{g\_scan} = C_f(2\frac{n}{p} + p - 4) + \sum_{i=1}^{p-1} T_{com}^{i\frac{n}{p}m}. \tag{6.46}$$

6.2.3.2 scan on the Hypercube

Algorithm PAR_SCAN (hypercube)
BEGIN

1. FOR every processor$_i$ DO in PARALLEL
   result $\longleftarrow$ **scan** f a xs_i
   last_result $\longleftarrow$ *last* **result**
2. FOR all processors DO
   FOR $j \longleftarrow 0$ to $(d-1)$ DO
      IF (bit$_j$ of processor_number $= 0$) THEN
         send **last_result** to neighbour in dimension j
      ELSE
         receive **data** from neighbour in dimension j
         last_result $\longleftarrow$ f **data last_result**
         store **data** in **data_list**
3. FOR each subcube$_i$ of dimension > 0 DO
   communicate partial results from lower-numbered
   subcubes to lower-numbered processors in subcube$_i$
4. FOR all processors DO
   par_result $\longleftarrow$ **fold** f a **data_list**
   /* *data_list has the partial results*
   from lower-numbered processors */
   FOR $j \longleftarrow 1$ to $(\frac{n}{p} - 1)$ DO
      result$_j$ $\longleftarrow$ f **par_result result**$_j$
   result$_{\frac{n}{p}}$ $\longleftarrow$ **last_result**

END.

Note that step 3 in the algorithm would involve at most $(d-1)$ communications. The cost expressions are derived in the following manner.

- **s_scan** - Adding the costs for each of the steps, in order,

$$\begin{aligned} C^h_{s\_scan} &= \underbrace{C_f(\frac{n}{p}-1)}_{step\_1} + \underbrace{d(C_f + T^1_{com})}_{step\_2} + \underbrace{T^1_{com}(d-1)}_{step\_3} + \\ & \underbrace{C_f(d-1) + C_f(\frac{n}{p}-2)}_{step\_4}. \end{aligned}$$

Grouping like terms provides the following:

$$C^h_{s\_scan} = C_f(2\frac{n}{p} + 2d - 4) + T^1_{com}(2d-1). \tag{6.47}$$

- **g_scan** - Representing the cost function by $g(l_1, l_2)$ gives:

$$\begin{aligned} C^{hf}_{g\_scan} &= \sum_{i=1}^{\frac{n}{p}-1} g(m, im) + \sum_{i=0}^{d-1} (T_{com}^{2^i \frac{n}{p} m} + g(2^i \frac{n}{p} m, 2^i \frac{n}{p} m)) + \\ & (d-1) T_{com}^{2^{d-1} \frac{n}{p} m} + \sum_{i=1}^{\frac{n}{p}-2} g((p-1)\frac{n}{p} m, im). \end{aligned} \tag{6.48}$$

The cost for the input-independent case is given by:

$$C^{hc}_{g\_scan} = C_f(2\frac{n}{p} + 2d - 4) + \sum_{i=0}^{d-1} T_{com}^{2^i \frac{n}{p} m} + (d-1) T_{com}^{2^{(d-1)} \frac{n}{p} m}. \tag{6.49}$$

6.2.3.3 scan on the 2-D Torus/Mesh

The algorithm is based on the one described in [Akl89].

- **s_scan** - If the wrap-around links are not used, then the cost is given by:

$$\begin{aligned} C1^m_{s\_scan} &= \underbrace{C_f(\frac{n}{p}-1)}_{step\_1} + \underbrace{(T^1_{com} + C_f)(p_1 - 1)}_{step\_2} + \\ & \underbrace{(C_f(p_2-1) + T^1_{com}(p_2-1) + T^1_{com}(p_1-1))}_{step\_3} + \underbrace{C_f(1 + \frac{n}{p} - 1)}_{step\_4}. \end{aligned}$$

The final cost expression is given by:

$$C1^m_{s\_scan} = C_f(2\frac{n}{p} + p_1 + p_2 - 3) + T^1_{com}(2p_1 + p_2 - 3). \tag{6.50}$$

Algorithm <u>PAR_SCAN (mesh)</u>
BEGIN

1. FOR every processor$_i$ DO in PARALLEL
   result $\longleftarrow$ **scan** f a xs_i
   last_result $\longleftarrow$ *last* **result**
2. /* *update along the rows in parallel* */
   FOR all processors DO
   IF (not first-column processor) DO
   receive **data**$_1$ from left neighbour
   last_result $\longleftarrow$ f **data**$_1$ **last_result**
   forward **last_result** to right neighbour
3. FOR processors in last column DO
   /* *send partial results of each row to subsequent row* */
   IF (not first-row processor) DO
   receive **data**$_2$ from neighbour in previous row
   last_result $\longleftarrow$ f **data**$_2$ **last_result**
   forward **last_result** to neighbour in next row
   IF (not first-row processor) DO
   forward **data**$_2$ to left neighbour
   FOR all other processors DO
   IF (not first-row processor) DO
   receive **data**$_2$ from right neighbour
   forward **data**$_2$ to left neighbour
   /* *except first-column processors* */
4. FOR all processors DO
   data $\longleftarrow$ f data$_2$ data$_1$
   FOR $j \longleftarrow 1$ to $(\frac{n}{p} - 1)$ DO
   result$_j$ $\longleftarrow$ f **data result**$_j$

END.

- In step 3, instead of sending the partial results from the previous rows all along the length of the row, the wrap-around could be used to communicate it to the neighbour in the first column. Then, both processors could communicate the partial results in opposite directions along the row, thereby reducing the communication costs. Note that processors in the first row do not have to communicate partial results to neighbours in that row. So, even in the case of the non-uniform mesh, the cost can be improved to:

$$C^{m}_{s\_scan} = C_f(2\frac{n}{p} + p_1 + p_2 - 4) + T^1_{com}(\frac{3}{2}p_1 + p_2 - 1). \qquad (6.51)$$

- **g_scan** - With wrap-around links and a cost function $g(l_1, l_2)$:

$$\begin{aligned} C^{mf}_{g\_scan} &= \sum_{i=1}^{\frac{n}{p}-1} g(im, m) + \sum_{i=1}^{p_1-1} (T_{com}^{i\frac{n}{p}m} + g(i\frac{n}{p}m, \frac{n}{p}m)) + \\ &\sum_{i=1}^{p_2-1} (T_{com}^{i\frac{n}{p}mp_1} + g(i\frac{n}{p}mp_1, \frac{n}{p}mp_1)) + \\ &\frac{p_1}{2} T_{com}^{\frac{n}{p}mp_1(p_2-1)} + g(\frac{n}{p}mp_1(p_2-1), \frac{n}{p}m(p_1-1)) + \\ &\sum_{i=1}^{\frac{n}{p}-1} g(\frac{n}{p}m(p-1), im) \end{aligned} \qquad (6.52)$$

For the input-independent argument function the cost simplifies to:

$$\begin{aligned} C^{mc}_{g\_scan} &= C_f(2\frac{n}{p} + p_1 + p_2 - 3) + \sum_{i=1}^{p_1-1} T_{com}^{i\frac{n}{p}m} + \\ &\sum_{i=1}^{p_2-1} T_{com}^{i\frac{n}{p}mp_1} + \frac{p_1}{2} T_{com}^{\frac{n}{p}mp_1(p_2-1)}. \end{aligned} \qquad (6.53)$$

6.2.3.4 scan on the s-ary Tree

The parallel implementation on an s-ary tree is a modification of the version described in [Ble89], for binary trees. Assumption 4 requires modification for this algorithm. The elements of the input list are initially at the leaves of the tree. Any communication costs incurred in achieving this distribution will also be accounted for when the cost for the entire problem is computed.

$xs = xs_0 @ xs_1 @ \ldots @ xs_{\frac{n}{s^d}-1}$

where, $xs_i \in leaf_i$; $\qquad i \in 0, 1, \ldots, (n-1)$

Leaves are numbered from left to right. A tree of arity s and a depth d has s^d leaves. Therefore, $\mid xs_i \mid = \frac{n}{s^d}$.

```
Algorithm PAR_SCAN (s-ary tree)
BEGIN
      1. FOR every leaf processor DO in PARALLEL
              result ⟵ scan f a xs_i
              last_result ⟵ last result
              send last_result to parent
      2. FOR all non-leaf processors DO in PARALLEL
              FOR j ⟵ 0 to (s − 1) DO
                    receive data from child_j
                    temp_j ⟵ data
              result ⟵ scan f a temp
              IF (root processor) THEN
                 FOR j ⟵ 0 to (s-1) DO
                 send result_j to child_j
              ELSE
                    send (last result) to parent
      3. FOR all processors, except root and leaves DO
              receive data from parent
              send data to child_0
              FOR j ⟵ 1 to (s − 1) DO
                    temp_j ⟵ f data result_{j−1}
                    send result_j to child_j
      4. Leaf processors DO
              receive data from parent
              result_0 ⟵ data
              FOR j ⟵ 1 to (n/s^d − 1) DO
                    result_j ⟵ f data result_j
      5. Root processor_0 DO
              send (last result) to rightmost leaf
END.
```

Step 5 in the algorithm is essential since the last element of the resulting list will remain at the root. In an actual implementation, the last element can be communicated along with $\mathbf{temp}_{s-1}$ in step 3 of the algorithm. This will not increase the communication cost very significantly. In the following cost computations, Step 5 is assumed to be combined with Step 3, and the cost of communicating the additional element is ignored, to simplify the cost expression. After the **scan** operation the result is left distributed across the leaves of the tree.

Given a tree of depth d, the results take d steps to reach the root in step 2, and a further d steps for the results to reach the leaves again in step 4, with each step involving s communications.

- **s_scan**

$$\begin{aligned} C^t_{s\_scan} &= \underbrace{C_f(\frac{n}{s^d} - 1)}_{step\_1} + \underbrace{d(sT^1_{com} + C_f(s-1))}_{step\_2} + \\ &\quad \underbrace{((d-1)(s-1)C_f + dsT^1_{com})}_{step\_3} + \underbrace{C_f(\frac{n}{s^d} - 1)}_{step\_4} \end{aligned}$$

$$C^t_{s\_scan} = C_f(2\frac{n}{s^d} + (s-1)(2d+1)) + T^1_{com}2ds. \tag{6.54}$$

- **g_scan** - Different data sizes will be communicated in parallel on different branches of the tree during the down-sweep (i.e. steps 3 and 4). The largest data size is communicated along the rightmost branch of the tree, since the partial results from all the nodes to its left must reach the nodes on this branch. Also, the size of the data communicated increases with each level down the tree. The non-uniformity in data transfers results in a more complicated cost expression.

If $k = (s-1)s^{d-1}\frac{n}{s^d}m$ then

$$\begin{aligned} C^{tf}_{g\_scan} \quad = \quad & \underbrace{\sum_{i=1}^{\frac{n}{s^d}-1} g(im, m)}_{step\_1} + \underbrace{\sum_{i=0}^{d-1} sT_{com}^{s^i \frac{n}{s^d}m} + \sum_{i=0}^{d-1}\sum_{j=1}^{s-1} g(js^i\frac{n}{s^d}m, s^i\frac{n}{s^d}m)}_{step\_2} + \\ & \underbrace{\sum_{j=0}^{s-1} T_{com}^{js^{d-1}\frac{n}{s^d}m} + (s-1)\sum_{i=0}^{d-2} g(k, s^i\frac{n}{s^d}m) + T^k_{com}}_{step\_3} + \\ & \underbrace{s(T_{com}^{k+\frac{n}{s^d}ms^{d-2}} + T_{com}^{k+\frac{n}{s^d}m(s^{d-2}+s^{d-3})} + \ldots + T_{com}^{k+\frac{n}{s^d}m\sum_{i=1}^{d-2}s^i})}_{step\_3} + \\ & \underbrace{(s-1)T_{com}^{k+\frac{n}{s^d}m\sum_{i=0}^{d-2}s^i}}_{step\_3} + \underbrace{\sum_{i=1}^{\frac{n}{s^d}-1} g(k + \frac{n}{s^d}m\sum_{j=0}^{d-2}s^j, im)}_{step\_4}. \end{aligned}$$

The first three terms constitute the cost for the up-sweep and are easily deduced. The cost for the down-sweep is deduced as follows. At the root, the identity element is communicated to the leftmost child, data of size $s^{d-1}\frac{n}{s^d}m$ (which is the size of the first partial result) is communicated to child$_1$, …, data of size $(s-1)s^{d-1}\frac{n}{s^d}m$ is communicated to child$_{s-1}$. The cost of communicating the identity element can be assumed to be zero, since it does not actually have to be communicated. These communications account for the fourth term in the cost expression. At each level in the tree, there are $(s-1)$ applications of f performed in parallel. However, since the maximum data size is handled by the rightmost node at each level, the cost for that level will be determined by the cost of this node. Since there are $(d-1)$ non-leaf nodes (excluding root) on each branch in a tree of depth d, the cost for the $(s-1)$ applications of f is given by the fifth term in the expression. The terms that follow represent the communication costs. They account for the cost of communicating the increasing data sizes to the child nodes, from the rightmost node at each level, since largest sizes are communicated from the rightmost node. The last expression accounts for the update performed at the leaves of the tree, after the down-sweep. Once again, the cost is based on the rightmost leaf, since it would handle the largest data size. In practice, the implementation cost could be reduced if the parent at each level started by sending the partial results to the rightmost child.

6.2.4 filter

From the definition of **filter**, it is clear that the size of the output list is data-dependent. After this operation, the sizes of the sublists on different processors may be different. The exact nature of the output distribution is not predictable. However, since all the subsequent phases of the problem would assume balanced load, it appears that a load balancing operation needs to be performed after a phase containing the **filter** function. This would involve gathering the results from all the processors and redistributing them (if necessary), so that the load is balanced for the next phase. This may prove to be unnecessary if the **filter** operation results in balanced load for some data set. On the other hand, there may be extreme cases in which the output sublists on some processors are of the same size as the corresponding input sublists, (i.e. all the elements retained), but others in which the output sublists are empty. In the latter case, data redistribution is crucial to obtain good performance.

The sizes involved in gathering and redistributing the data are also unknown. The analyser, therefore, adopts a pessimistic approach in computing the cost of **filter** and assumes that the size of the output list is equal to the size of the input list. If there are several phases following the one involving the **filter** function, then the predicted costs may be much worse than the actual costs, and this in turn is dependent on the nature of the data set.

However, since the pessimistic approach assumes the availability of the entire data set after the **filter** operation, it may, in the worst case, result in the selection of a parallel implementation when a sequential one would have proved to be more cost-effective. A pessimistic approach to cost computation may result in an overly-optimistic approach to the selection of a parallel implementation. The incorporation of a profiler would alleviate this problem.

The sequential cost of **filter** is given by:

$$C^s_{filter} = C_f n. \tag{6.55}$$

The parallel implementation of **filter** on all the topologies is the same, except that each uses its own *gather* algorithm.

```
Algorithm PAR_FILTER (all topologies)
BEGIN
        1. FOR every processor_i DO in PARALLEL
                result <--- filter f xs_i
        2. gather result from all processors
END.
```

The cost of the parallel implementation is given by:

$$C^p_{filter} = C_f \frac{n}{p} + G \tag{6.56}$$

where G is the cost of gathering n elements distributed across p processors on the particular topology. The cost of redistributing the result of the **filter** operation is not accounted for in its cost, but will be, by the subsequent phase. This will be zero if the latter is sequential.

6.2.5 inits and tails

For an input list of size n, the function **inits** produces $(n + 1)$ sublists that successively increase in size. The size of sublist$_i$ is *one* more than the size of

sublist_{i-1}. The total number of elements in all the sublists is given by:

$$\begin{aligned} total &= 0 + 1 + 2 + \cdots + n \\ total &= \frac{n(n+1)}{2}. \end{aligned}$$

The computation of the function **inits** only involves list processing costs. In a parallel implementation of **inits**, the communication cost would make a significant contribution to the total cost. As discussed in Section 4.1, **inits** can be expressed in terms of the function **scan** and can, therefore, be implemented in parallel by using the parallel **scan** algorithm. However, this is not desirable for two reasons.

- Since **inits** produces sublists of increasing sizes, the load on the processors would be imbalanced after the **inits** operation, even if there were the same number of elements on all the processors initially. This would cause non-uniform processor utilisation for subsequent phases of the problem.

- During the implementation of the **scan** algorithm, the partial results exchanged between processors would be of unequal sizes. The size of the data communicated increases with the progress of the algorithm. This is undesirable, and it can be shown that it leads to higher communication costs compared to the alternative suggested below.

In order to obtain balanced load after the **inits** operation, ideally, the total number of elements from all the sublists on each processor should be $\frac{n(n+1)}{2p}$. This is achieved by allowing different processors to have different numbers of sublists. Some processors have a larger number of sublists of smaller sizes, while others have a smaller number of sublists of larger sizes. Each processor calculates the number of sublists that will be resident on it after the **inits** operation. In the following discussion, the *nil* list which is the first element of the result of **inits** is ignored.

After the **inits** operation, the number of sublists on processor$_i$ is represented by n_i. In particular, on processor$_0$ the number of sublists is n_0, and the length of sublist$_i$ is i. The condition which has to be satisfied in order to obtain balanced load is as follows:

Total number of elements in all sublists on processor$_0$ $= \frac{n(n+1)}{2p}$.
$\Rightarrow 1 + 2 + 3 + \cdots + n_0 = \frac{n(n+1)}{2p}$
$\Rightarrow \frac{n_0(n_0+1)}{2} = \frac{n(n+1)}{2p}$
Solving for n_0 gives,

$$n_0 = \frac{-1 + \sqrt{1 + \frac{4n(n+1)}{p}}}{2}. \tag{6.57}$$

The last sublist on processor$_0$ is of length n_0 and consequently, the first sublist on processor$_1$ is of length $n_0 + 1$. Again, the condition which has to be satisfied to obtain a balanced load results in the following deduction.

Total number of elements in all sublists on processor$_1$ $= \frac{n(n+1)}{2p}$.
$\Rightarrow (n_0 + 1) + (n_0 + 2) + \cdots + (n_0 + x) = \frac{n(n+1)}{2p}$
$\Rightarrow x = \frac{\sqrt{(2n_0+1)^2 + \frac{4n(n+1)}{p}} - (2n_0+1)}{2}$

This could be generalised to the following.

For every processor$_i$, where $i = 1, 2, \ldots, (p-1)$

length of first sublist $= n_{i-1} + 1$

length of last sublist $= n_{i-1} + x$
where, $x = \frac{\sqrt{(2n_{i-1}+1)^2 + \frac{4n(n+1)}{p}} - (2n_{i-1}+1)}{2}$

and

length of first sublist on processor$_0$ $= 1$

length of last sublist on processor$_0$ $= n_0$ (Equation 6.57)

Each processor first determines the number and sizes of the sublists that it should generate, which is done locally. This implies that each processor should possess a copy of the input list. If the input list is not already resident on all the processors, then it should be broadcast to them. If it is already scattered across the network, then a total exchange operation would have to be performed, so that each processor obtains a copy of the entire list. If C_{be} represents the cost of Broadcast or Total Exchange, the cost of **inits** is given by:

$$C^p_{inits} = C_{be} \tag{6.58}$$

List processing costs involved in generating the sublists are not accounted for in the cost expression since the analyser accounts for such costs for every output list that is constructed. A similar strategy is applied to **tails**.

6.2.6 cross_product

The costs for both **r_cross_product** and **c_cross_product** are identical and their costs are therefore discussed jointly. Since **cross_product** has two argument lists, an additional assumption for cost calculations is required.

The second input list is represented by ys

where, $\mid ys \mid = m$ and $ys \in \text{processor}_0$

The second list is assumed to be resident on a single processor, so the cost for a parallel implementation must account for the cost of distributing this list to the other processors. The sequential cost of **cross_product** is given by:

$$C^s_{cross\_product} = C_f nm \tag{6.59}$$

Algorithm <u>PAR_CROSS_PRODUCT (all topologies)</u>
BEGIN
  Broadcast ys to all the processors
  FOR every processor$_i$ DO in PARALLEL
    result $\longleftarrow$ **cross_product** $f\ xs_i\ ys$
END.

From the definitions of **r_cross_product** and **c_cross_product**, it is clear that if the first input list is distributed across p processors in the network, then the second list would need to be broadcast to all the processors in order to evaluate the **cross_product** in parallel. The parallel implementations are identical on all the topologies, except for the different algorithms and the corresponding costs involved in the broadcast operation. The results are left distributed across the network.

The cost is given by:

$$C^{p}_{cross\_product} = B + C_f \frac{n}{p} m \tag{6.60}$$

where, B is the cost of broadcasting m elements to p processors, on the particular topology.

6.2.7 composition

It is clear from the definition of **composition** that each phase uses the results of the previous phase. A possible way of implementing **composition** in parallel would be to pipeline the intermediate results, so that the implementation of the next phase could be overlapped with that of the current phase [Kel89]. However, in the current scheme, **composition** is implemented sequentially. Any potential parallelism is exploited within each phase of the **composition**.

The cost of a **composition** involving k phases is, therefore, given by:

$$C_{composition} = \sum_{i=1}^{k} C_i \tag{6.61}$$

where, C_i is the cost of phase$_i$, together with any data rearrangement costs that would be incurred in its implementation.

6.2.8 map2 and zip

map2 is just an extension of **map** to cater for two argument lists. If both the input lists are already distributed across the network, then the cost of **map2** is identical to the cost of **map**. If not, then the additional cost of scattering the second list on the relevant processor network must be added to the cost of **map**. A similar strategy is followed for the parallel implementation of **zip**.

6.2.9 The iterative functions

As pointed out earlier, none of the iterative functions are themselves implemented in parallel. If the argument of an iterative function comprises of a composition of recognised functions, then the composition could be implemented in parallel using the analysis described in Section 5.2. The cost of an iterative function is taken to be the cost of a *single* application of its argument function *f*. An implicit assumption here is that the costs of all the iterations are equal. Since the strategy for the selection of a parallel implementation relies only on cost comparisons, the absolute cost of all the iterations is not required. Moreover, in many cases, e.g. **iterate_cond**, it may not be possible to determine the number of iterations at compile-time.

At the end of each iteration, the results are left distributed. If the implementation of the next iteration requires the data to be distributed in the same way as left by the previous phase, then no data rearrangement cost is incurred. If not, the cost of data rearrangement is added to the cost of the iterative function.

In particular, in the case of a parallel implementation of **iterate_cond**, the result is left distributed across the processor network for the implementation of the next iteration. The *condition* is only evaluated on a *copy* of the result. This could save on potential communications overheads.

Again, there are limitations that arise as a result of assuming identical cost for every iteration. The model attempts to minimise potential discrepancies in the estimated costs, e.g. initial distribution costs at the start of the first iteration are not added to the cost of the iterative function if these costs will not be incurred by subsequent iterations. The idea is that such a cost will be negligible when averaged over a large number of iterations (assuming there are a fairly large number of iterations). On the other hand, if this initial distribution cost is added, then it will substantially increase the estimated implementation cost and may lead to the selection of a poor parallel implementation.

6.2.10 split

The introduction of **split** as a recognised function was motivated by the need to express divide-and-conquer type of applications. The function **split** takes two arguments, an integer $k > 0$ and a list of size n. The result of applying the function to the list is to split the list into k sublists, with $(k-1)$ sublists of size $\lceil \frac{n}{k} \rceil$ and the last sublist of length $n - (k-1)\lceil \frac{n}{k} \rceil$. **split** only involves list rearrangement costs which are computed by the analyser, based on the nature of the input list. The cost is represented by:

$$C^s_{split} = S_{n,k}. \tag{6.62}$$

For a parallel implementation of **split** with the list distributed across p processors, each processor splits the part of the list resident on it into $\lceil \frac{k}{p} \rceil$ parts. If k is not divisible by p, then the parallel implementation would result in the list being split into a slightly different number of sublists as compared to a sequential implementation for the same list. If it is the case that this would not affect the outcome of the final result itself, then this difference may be acceptable. An example is the merge sort program described in Section 4.3.2. Since the combining function *merge* is associative, slight differences in the manner in which the list is split would not affect the result. However, there may be cases where this might produce erroneous results. In such cases, the programmer is invited to use a variant of the function **split**, called **seq_split**. Its definition is identical to that of **split**, but it would force the analyser (and in turn the code generator) to consider only a sequential implementation for **split**.

```
Algorithm PAR_SPLIT (all topologies)
BEGIN
        FOR every processor_i DO
              result <-- split k/p xs_i
END.
```

Adopting the terminology mentioned earlier, the cost of the parallel implementation of **split** can be represented by:

$$C^{p}_{split} = S_{\frac{n}{p},\frac{k}{p}} \tag{6.63}$$

If the value of k is unknown at compile-time, then the analyser assumes that it is equal to n. In other words, each sublist is assumed to be of size *one*. This again, is a pessimistic approach to cost computation, and could result in the selection of a poor parallel implementation. Again, profiling techniques could help in dealing with the problem.

6.2.11 The Combinator $\mathcal{R}^k$

The definition for $\mathcal{R}^2$, the case where a recognised function operates on two input lists, is as follows (refer to Section 4.2):

$\mathcal{R}^2$ F f_1 f_2 ... f_k g h xs = F f_1 f_2 ... f_k (g xs) (h xs)

where, F is a recognised function with 2 input lists

f_1, f_2, ..., f_k are parameters to F

g and h are functions that operate on lists.

In general, g and h could be any complex functions including other recognised functions. However, some situations may just require two copies of the same list, in which case both of these functions would be the identity function.

The present set of recognised functions have, at most, two argument lists, so the following discussion focuses on $\mathcal{R}^2$. The number of possibilities that arise in the selection of a parallel implementation of $\mathcal{R}^k$ makes it a complex scheduling problem in itself and will not be discussed here. In the case of $\mathcal{R}^2$, the discussed heuristic is used to prune the search-space, while attempting not to eliminate implementations that may eventually result in the least cost. This cannot, however, be guaranteed and in some cases the least-cost implementation may be eliminated, since the heuristic performs cost optimisation that is local to the phase under consideration. The heuristic computes the costs for the various possible implementations which are considered. The least-cost one is then selected.

C^s_g represents the sequential cost of g

$C^{p'}_g$ represents the cost of the parallel implementation of g on p' processors

$MAX(x, y)$ = if $(x > y)$ then x else y

$MIN(x, y)$ = if $(x < y)$ then x else y

$C^{p'}_B$ is the cost of broadcasting the list to p' processors in the topology

$C^{p'}_{GS}$ is the cost of gathering/scattering the list from/to p' processors in the topology.

HEURISTIC PAR_$\mathcal{R}^2$
BEGIN

1. IF g and h are sequential functions THEN
   Gather list from p processors
   a. Implement g and h on one processor
      $C_1 \longleftarrow C^p_{GS} + C^s_g + C^s_h$
   b. Send copy of list to Processor_1
      Implement g on Processor_0
      Implement h on Processor_1
      $C_2 \longleftarrow C^p_{GS} + MAX(C^s_g, C^s_h) + C^2_B$
   Implement F in parallel on p processors
   $C_{R1} \longleftarrow C^p_F + C_1 +$ **data-rearrangement-costs**
   $C_{R2} \longleftarrow C^p_F + C_2 +$ **data-rearrangement-costs**
   $C_{\mathcal{R}} \longleftarrow MIN(C_{R1}, C_{R2})$
2. IF g and h are recognised functions THEN
   a. Compute C_{R1} and C_{R2} as in step 1
      $C_{Rp1} \longleftarrow MIN(C_{R1}, C_{R2})$
   b. /* *assuming two copies of list available* */
      Implement g in parallel on p processors
      Implement h in parallel on p processors
      $C_2 \longleftarrow C^p_g + C^p_h$
   Implement F in parallel on p processors
   $C_{Rp2} \longleftarrow C^p_F + C_2$
   $C_{\mathcal{R}} \longleftarrow MIN(C_{Rp1}, C_{Rp2})$
3. IF g is sequential and h is recognised THEN
   $C_{\mathcal{R}} \longleftarrow C^p_{GS} + C^s_g + C^p_h + C^p_{GS} + C^p_F$
4. OUTPUT C_R

END

The implementation heuristic for $\mathcal{R}^2$ follows the usual rules. If g and h are sequential functions and the input list is distributed, then the list has to be gathered before g and h can be evaluated. The two transformed lists can then be redistributed, depending on whether or not a parallel implementation is selected for F. If g and/or h involve recognised functions, then a distributed implementation could be selected for g and/or h, so that it results in the output data being distributed as required by F. This would save data rearrangement overheads. In any case, the resulting distribution of the two lists must be such that it is suitable for the implementation selected for F and any cost that is incurred in achieving this distribution is represented by the term, **data-rearrangement-costs**, in the heuristic.

6.2.12 get_neigh

The sequential implementation of **get_neigh** only involves list rearrangement and so the only cost of implementation comprises of list processing costs. If this cost is represented by G_n, then

$$C^{s}_{get\_neigh} = G_n. \tag{6.64}$$

The algorithms for the parallel implementation of **get_neigh** on the various topologies do not correspond to its definition in terms of the basic set of recognised functions. It may be observed that only the first and the last elements of (the part of) the list resident on each processor have non-local neighbours. Each processor is, therefore, involved in a communication routine that obtains these values. Then all the processors can execute a local **get_neigh** operation in parallel to obtain the final result, which is distributed across the network. In the following algorithms the functions *first* and *last* return the first and last elements of a list, respectively.

6.2.12.1 get_neigh on the Linear Array

Algorithm PAR_GET_NEIGH (Linear Array)
BEGIN
1. FOR every processor DO
   result_left $\longleftarrow$ *first* xs_i
   result_right $\longleftarrow$ *last* xs_i
2. Odd-numbered processors DO
   send **result_left** to left neighbour
   receive **data_left** from left neighbour
   /* *this is the left neighbour for first element* */
   send **result_right** to right neighbour
   receive **data_right** from right neighbour
   /* *this is the right neighbour for last element* */
   Even-numbered processors DO
   receive **data_right** from right neighbour
   send **result_right** to right neighbour
   Except processor$_0$ DO
   receive **data_left** from left neighbour
   send **result_left** to left neighbour
3. FOR every processor DO
   /* *perform* **get_neigh** *locally* */
   result $\longleftarrow$ **get_neigh data_left data_right** xs_i
END.

The algorithm comprises of two communication steps, and in each of them one list element is exchanged between pairs of processors. The total cost is:

$$C^a_{get\_neigh} = 4T^1_{com} + G_{\frac{n}{p}}. \tag{6.65}$$

6.2.12.2 get_neigh on the Hypercube

The described algorithm implements **get_neigh** in parallel on a hypercube of dimension d with p processors. It comprises of three stages. The first stage comprises of $(d-1)$ communication steps in which lower-numbered processors forward their first and last elements, i.e. the boundary values, to higher-numbered ones. This happens in parallel in the two subcubes of dimension $(d-1)$. The second stage is the exchange step in which the boundary values in the two subcubes are exchanged. The third stage again comprises of $(d-1)$ communication steps in which the higher-numbered processors communicate the boundary values to the lower-numbered ones. At the end of $2d$ steps, every processor will have obtained the neighbouring values for all its local elements. All the processors can then execute **get_neigh** locally to produce the result. The following notation is used in the algorithm.

p_no is the processor-number in binary representation; $0 \leq p\_no < p$
$neigh\_no_i$ is the number of the neighbouring processor in dimension i,
in binary representation
$is\_one(i, p\_no)$ returns TRUE if the last i bits of p_no are 1, FALSE otherwise
T is a 2-element array on each processor

If the cost of the initialisation step is ignored, then the cost of the algorithm for **get_neigh** is given by:

$$C^h_{get\_neigh} = T^2_{com}(d-1) + 2T^2_{com} + T^2_{com}(d-1) + G_{\frac{n}{p}}.$$

On simplification, the cost is given by:

$$C^h_{get\_neigh} = T^2_{com}2d + G_{\frac{n}{p}}. \tag{6.66}$$

The described algorithm results in a better implementation when compared to the one corresponding to gathering the list, executing **get_neigh** on a single processor and then redistributing the resulting list. The latter procedure would involve a *gather* operation, followed by an implementation of **get_neigh** on a single processor, followed by a *scatter* operation. The total cost in that case would be given by:

$$Cost^n_{get\_neigh} = 2(dK_0 + \frac{1}{K_1}\frac{n}{p}(p-1)) + G_n. \tag{6.67}$$

Algorithm PAR_GET_NEIGH (hypercube)
BEGIN
    1. FOR every processor DO
        /* *initialisation step* */
        IF p_no is even THEN $M = 0$
        ELSE $M \longleftarrow$ largest i s.t., $is\_one\ (i, p\_no)$ = TRUE;
                $0 < i < d$
        let $q_0, q_1, \ldots, q_M$ be 2-element queues
        insert *first* xs_i into q_0
        insert *last* xs_i into q_M
    2. FOR $i \longleftarrow 0$ TO $(d-2)$ DO
        /* *stage 1 - forward communication* */
        IF $(is\_one\ (i, p\_no))$ THEN
            IF $(p\_no < neigh\_no_i)$ THEN
                Send q_i to $neigh\_no_i$
                $q_i \longleftarrow$ NIL
            ELSE
                $T \longleftarrow$ data received from $neigh\_no_i$
                IF $(p\_no - neigh\_no_i = 1)$ THEN
                    /* *neighbour for first element received* */
                    insert $T[1]$ into **ln** list
                    insert $T[0]$ into q_{i+1}
                ELSE insert $T[0]$ into q_{i-1}, $T[1]$ into q_{i+1}
    /* CONTD ... */

```
Algorithm PAR_GET_NEIGH (hypercube)
CONTD
        3. IF (is_one (i(= d - 1), p_no)) THEN
               /* Stage 2 - Exchange step */
               IF (p_no < neigh_no_i) THEN
                  Send q_i to neigh_no_i
                  T <-- data received from neigh_no_i
               ELSE
                  T <-- data received from neigh_no_i
                  Send q_i to neigh_no_i
               IF (i = 0) THEN        /* 2-D hypercube */
                  insert T[0] and T[1] into ln list
               ELSE
                  insert T[0] into q_{i-1}
                  insert T[1] into ln list
        4. FOR i <-- (d - 2) DOWNTO 0 DO
                  /* stage 3 - Backward Communications */
                  IF (is_one (i, p_no)) THEN
                     IF (p_no < neigh_no_i) THEN
                        T <-- data received from neigh_no_i
                        insert T[0] into ln list
                        IF (i > 0) THEN
                           insert T[1] into q_{i-1}
                        ELSE insert T[1] into ln list
                     ELSE
                        Send q_i to neigh_no_i
                        q_i <-- NIL
        5. FOR every processor DO
                  result <-- get_neigh ln[0] ln[1] xs_i
END.
```

In Equation 6.66, $T^2_{com} = K_0 + 2\frac{1}{K_1}$ (See Equation 6.20). Clearly, for any value of $n > 2$, Equation 6.66 corresponds to a lower implementation cost as compared to Equation 6.67. The algorithm does not result in an order of magnitude decrease in cost. However, the implementation corresponding to Equation 6.67 would also suffer from much higher list processing costs, since $G_n > G_{\frac{n}{p}}$. The communication costs in that case would also be greater, since the size of the resulting list involved in the scatter operation is of a larger size than the input list.

6.2.12.3 get_neigh on the 2-D Torus/Mesh

```
Algorithm PAR_GET_NEIGH (2-D Torus/Mesh)
BEGIN
        1. /* identical to PAR_GET_NEIGH (Linear Array) */
        2. /* identical to PAR_GET_NEIGH (Linear Array) */
        3. First Column Processors DO
                 Send data_left to south neighbour
           Last Column Processors DO
                 Send data_right to north neighbour
        4. FOR every processor DO
                 result <-- get_neigh data_left data_right xs_i
END.
```

The first two steps of the algorithm are identical to that of the linear array. In other words, each row of the torus is treated as a linear array, except for the wrap-around link. The processors in the first column treat the ones in the last column (and the same row) as their left neighbours and vice-versa. After the second step, each processor in the first column still requires the left neighbour for its first element and each processor in the last column requires the right neighbour for its last element. In the third step, processors in the first column obtain this value from their neighbour above and processors in the last column obtain it from their neighbour below. Steps 1 and 2 in the algorithm are identical to the ones in PAR_GET_NEIGH (Linear Array). The communication cost in Step 3 can be reduced by nearly half, by overlapping the *sends* in the two halves of the columns. The cost of the algorithm is then given by:

$$C^{m}_{get\_neigh} = T^{1}_{com}(\frac{p_2}{2} + 4) + G_{\frac{n}{p}}. \tag{6.68}$$

6.2.12.4 get_neigh on the Tree

```
Algorithm PAR_GET_NEIGH (s-ary Tree)
BEGIN
    1. FOR every Processor DO
           Create Queues q_up and q_d1, q_d2, ..., q_ds
    2. Leaf Processors DO
           insert first xs_i and last xs_i into q_up
       Non-leaf Processors DO
           insert first xs_i into q_up; last xs_i into q_d1
    3. /* Up-sweep */
       Leaf Processors DO
           Send q_up to parent
       Non-Leaf Processors DO
           FOR i <-- 1 to s DO
               Receive q from child_i
               IF (i = 1) THEN
                   /* received right neigh for last elm */
                   Insert q[1] into local_neigh list
                   q_d(i+1) <-- q[2]
               ELSE
                   q_d(i-1) <-- q[1]
                   IF (i + 1) <= s THEN
                       q_d(i+1) <-- q[2]
                   ELSE
                       q_up <-- q[2]
    /* CONTD ... */
```

Algorithm PAR_GET_NEIGH (s-ary Tree)
CONTD
4. /* *Down-sweep* */
Root Processor DO
Send q_{d_i} to child$_i$
Non-Leaf Processors DO
Receive q from parent
/* *received left neighbour for first elm* */
Insert $q[1]$ into **local_neigh** list
IF two elements received THEN
/* *processors in rightmost branch receive 1 value* */
$q_{d_s} \longleftarrow q[2]$
Send q_{d_i} to child$_i$
Leaf Processors DO
Receive q from parent
/* *left neigh for first elm is* $q[1]$ */
/* *right neigh for last elm is* $q[2]$ */
local_neigh $\longleftarrow q$
5. FOR all processors DO
result $\longleftarrow$ **get_neigh local_neigh[1]**
local_neigh[2] xs_i
END.

The algorithm comprises of two sweeps - an up-sweep in which processors send the elements which are meant for processors in the sub-trees to their left or right, to their respective parents; and a down-sweep in which the parents forward the acquired data to their respective children. Queues, q_{up}, to send data to parent and $q_{d_1}, q_{d_2}, \ldots, q_{d_s}$ to send data to child$_i$ $(1 \leq i \leq s)$ respectively, are maintained by each processor. Obviously, the root processor does not need to maintain q_{up} and similarly, the leaf processors do not require queues q_{d_i}, but these are not explicitly mentioned in the algorithm. Steps 3 and 4 of the algorithm each involve s communications of two elements. For a tree of depth d, there are d such communications in each of steps 3 and 4, respectively. The cost is given by:

$$C^t_{get\_neigh} = T^2_{com}2ds + G_{\frac{n}{p}}. \tag{6.69}$$

6.2.13 len

The sequential cost of **len** is straightforward.

$$C^s_{len} = C_f(n-1) \tag{6.70}$$

where, C_f in this case is the cost of a '+' operation.

The parallel implementation of **len** corresponds to those of its composing functions, viz. **map** and **fold** (refer to Section 4.2). In the following algorithm, PAR_FOLD represents the parallel **fold** algorithm on the relevant processor topology.

Algorithm PAR_LEN (all topologies)
BEGIN
    1. FOR every processor$_i$ DO in PARALLEL
        temp $\longleftarrow$ **map** *subst_1* xs_i
    2. **result** $\longleftarrow$ PAR_FOLD *plus* 0 **temp**
END.

The cost of implementing **len** in parallel is given by:

$$C^p_{len} = C_{subst\_1}(\frac{n}{p} - 1) + C^{plus}_{PAR\_FOLD} \tag{6.71}$$

where, $C^{plus}_{PAR\_FOLD}$ is the cost of the parallel **fold** operation on the specific architecture with the sequential function *plus* as its argument.

It may prove to be expensive to implement **len** in parallel, since the arguments of the **map** and **fold** functions are trivial operations. The communication cost which is incurred in scattering the input list and in the implementation of **fold**, would probably make it more expensive to implement it in parallel than sequentially. However, if the elements of the input list are already scattered across the network, and the **len** function is encountered, then it may prove to be more expensive to gather the elements of the list for the purpose of merely computing its length. In such cases, the parallel implementation will be chosen by the analyser.

6.2.14 select

The **select** function does not involve any computation. For a list data structure the sequential cost of **select** is proportional to the length of the list that needs to be traversed to locate the required element. In the worst case, this length would be equal to the size of the list. Since the index of the element to be selected may not be known until run-time, the analyser adopts a pessimistic approach in computing the cost of **select**. The sequential cost of **select** is, therefore:

$$C^s_{select} \propto n \tag{6.72}$$

The constant of proportionality is the cost of list processing that is estimated by the analyser.

select has been defined in terms of a **fold** operation (refer to Section 4.2). However, from the definition, it is clear that the argument of **fold** is not an associative function. Hence, the usual parallel implementation for **fold** cannot be applied. In a distributed implementation, the element to be selected would be resident on some processor. If the index of this element is unknown at compile-time, then this processor can only be identified at run-time, depending on the size of the input list and the index of the element. In the following algorithm, it is assumed that every processor holds the value of $\frac{n}{p}$ and that p_no represents the (unique) number of a processor.

Algorithm PAR_SELECT (all topologies)

/* *selects the j^{th} element of a distributed list* */

BEGIN

1. FOR every processor$_i$ DO in PARALLEL

   IF ($\frac{n}{p}p\_no \leq j < \frac{n}{p}p\_no + \frac{n}{p}$) THEN

   /* *this processor holds the required element* */

   select ($j - \frac{n}{p}p\_no$)

END.

The algorithm is very simple. It just accounts for the fact that the original input list is distributed across p processors and computes the offset from the actual value of j, to obtain the index value within a processor. If the index falls within the range of the indices of the elements that a particular processor holds, then it selects the element corresponding to that index. The selected element is left on the processor on which it was found. The cost of the parallel implementation is similar to the sequential one, but since each processor only holds a part of the entire list, the cost is given by:

$$C^p_{select} \propto \frac{n}{p} \tag{6.73}$$

6.2.15 apply_select

The cost of **apply_select** depends on the number of elements of the list to which the argument function f is applied. If this number is assumed to be m, then the sequential cost is given by:

$$\begin{aligned} C^s_{apply\_select} &= mC_f \\ & m \leq n \end{aligned} \tag{6.74}$$

If the value of m is unknown, then it is assumed that $m = n$. In other words, the worst-case cost is assumed.

The parallel implementation for **apply_select** on any topology is described by the following algorithm.

```
Algorithm PAR_APPLY_SELECT (all topologies)
      /* indices to be selected are i_j, i_k, ..., i_m */
BEGIN
      1. FOR every processor_i DO in PARALLEL
              FOR every index b DO
                  IF ((n/p)p_no <= b < (n/p)p_no + (n/p)) THEN
                      /* element_b ∈ processor_i */
                      element_b <-- f element_b
END.
```

The worst-case situation occurs if all the elements to which the function f is to be applied are resident on the same processor.

$$\begin{aligned} C^p_{apply\_select} &= mC_f \\ m &\leq \frac{n}{p} \end{aligned} \tag{6.75}$$

6.2.16 copy

copy is a communication construct. The cost of implementing **copy** sequentially follows from its definition and is proportional to the length of the list. It comprises of the list processing costs involved in selecting the appropriate element and then constructing a new list of pairs with the selected element and each of the other list elements. Since the index of the element to be copied may not be known until run-time, a worst-case situation is assumed, and the cost of **copy** is given by:

$$C^s_{copy} \propto 2n. \tag{6.76}$$

Algorithm PAR_COPY (all topologies)

/ copies the j^{th} list element to all other list elements */*

BEGIN

1. Call PAR_SELECT to select the j^{th} element
2. Processor on which the element was found DO
   Broadcast the element to all processors
3. Every Processor DO
   Pair up the received element with each list element

END.

The costs of steps 1 and 3 involve only list processing costs and are proportional to the length of the list that is resident on each processor, viz. $\frac{n}{p}$. If this cost is represented by C_{lproc} and the cost of broadcasting a single element of the list to all the processors is represented by B_1, then the cost of the parallel implementation of **copy** is given by:

$$C^p_{copy} = C_{lproc} + B_1. \tag{6.77}$$

6.2.17 reverse

The sequential implementation of **reverse** only involves list processing costs incurred in constructing the new (reversed) list. This cost is proportional to the length of the list:

$$C^s_{reverse} \propto n \tag{6.78}$$

The parallel implementation for **reverse** does not always follow its definition. If the entire list is resident on a single processor, then the usual sequential implementation would apply. If the list is distributed across several processors, then the following algorithms are adopted on the different topologies.

6.2.17.1 reverse on the Linear Array

There seems to be no efficient way of reversing a list that is distributed across a linear array. The parallel implementation in this case just follows the definition for **reverse** in terms of **g_fold**. So, the algorithm for **reverse** on the linear array is the same as that for **fold**, and the cost is given by the cost of **g_fold** on the linear array (Equation 6.27), with the sequential function being *rev*. (refer to Section 4.2). The resulting list would be left on processor$_0$.

6.2.17.2 reverse on the Hypercube

```
Algorithm PAR_REVERSE (hypercube)
BEGIN
        1. FOR every processor DO
                result ⟵ reverse xs_i
        2. FOR i ⟵ (d − 1) DOWNTO 0 DO
                exchange result with neighbour in dimension i
END.
```

The algorithm results in the reversed list being distributed across the processor network in a manner identical to the original list. The cost of reversing the portion of the list on each processor is, again, proportional to the length of the list that is resident on it. If this cost is represented by C_r, then the cost of the parallel implementation of **reverse** is given by:

$$C^h_{reverse} = C_r + 2T^{\frac{n}{p}}_{com} \tag{6.79}$$

The described algorithm provides a more economical implementation as compared to just gathering the list, reversing it and then re-distributing the reversed list. The number of communications required may not be very different in the two cases, but the amount of data involved in each communication step is much less in the recommended algorithm. Also, since the processors perform the **reverse** operation in parallel on different parts of the list, C_r would be less than $C^s_{reverse}$.

6.2.17.3 reverse on the 2-D Torus/Mesh

As in the case of the linear array, the algorithm for the parallel implementation of **reverse** on the 2-D torus follows its definition in terms of the basic set of recognised functions. The algorithm is the same as that for **fold** on the 2-D torus and the cost is given by Equation 6.33. The result is left on processor$_0$ and an additional scattering cost will be incurred if it needs to be redistributed for the subsequent phase.

6.2.17.4 reverse on the s-ary Tree

It appears to be difficult to design an efficient algorithm for reversing a list of elements distributed across a tree. The algorithm, therefore, simply follows the definition of **reverse** in terms of the basic set of recognised functions. The cost is given by Equation 6.39.

6.3 The 2-D Torus Topology

The cost expressions clearly indicate (not surprisingly!) that most of the recognised functions can be implemented most efficiently on the hypercube topology, with the 2-D torus proving to be the next-best candidate. The tree and the linear array prove to be inefficient and are not studied further. Chapter 7 presents the results of implementing three example programs on the hypercube topology. Although this thesis does not provide a practical implementation on the torus, a possible strategy is discussed in this section.

The hypercube is likely to have a much better performance than the 2-D torus in the context of the implementation strategy, since it is divisible into subcubes, in which the communication latencies are identical to that of the original hypercube. Two problems could arise as a result of dividing the torus into smaller parts:

- Logical neighbours in the smaller tori will not necessarily be physical neighbours. This would lead to an increase in communication costs, since a logical neighbour may no longer be just one hop away.
- Also, the numbering on the nodes in the smaller tori may not be as required for the parallel implementation of the particular phase. A rearrangement of the data may be necessary in order to obtain correct data placement on processors in the smaller tori. Not only would this contribute to the inter-phase data rearrangement costs, but it may also complicate the communication algorithms required.

In view of the problems just discussed, an alternative parallel implementation strategy could be considered for the 2-D torus. A hypercube embedding could be defined on the 2-D torus and the recognised functions could be implemented on this logical hypercube, using the corresponding hypercube algorithms. One example of an embedding of a hypercube in a 2-D torus is described in [Bre83]. In the following discussion, this logical hypercube is referred to as the *embedded hypercube*. The following points should be noted.

1. A hypercube of dimension d with $p = 2^d$ processors would be embedded in a 2-D torus with $p = p_1 p_2$ processors, where $p_1 = 2^{\lfloor \frac{d}{2} \rfloor}$ and $p_2 = 2^{\lceil \frac{d}{2} \rceil}$. p_1 represents the number of processors in each row and p_2 represents the number of processors in each column.

2. Logically neighbouring processors will not necessarily be physical neighbours in the embedded hypercube of dimension greater than four, since each processor in a 2-D torus has only four nearest neighbours. Nearest-neighbour preserving embeddings should be possible for hypercubes with $d \leq 4$.

3. The 2-dimensional hypercube is identical to the torus with four processors, including the numbering on the nodes. It could form the base case for an algorithm which would work by combining two $(d-1)$-dimensional embedded hypercubes, to construct the d-dimensional one.

Although the physical connectivity of the embedded hypercube is not as good as that of the corresponding hypercube, it may still prove to be more cost-effective than the corresponding 2-D torus, especially for small values of d. New data distribution algorithms, such as *scatter*, must be devised for the embedded hypercube and their costs computed. The costs of the recognised functions on the embedded hypercube must be computed, accounting for the additional hops required to communicate with logically neighbouring processors that are not physical neighbours.

A hypercube embedding appears to overcome some of the problems associated with the torus. However, rigorous cost computations both for the algorithms implementing the recognised functions and the communication routines on the embedded hypercube are required before a final conclusion can be reached. Other constraints which would affect the choice are the following:

- The number of processors in the network (p) - For small values of p, it may be more cost-effective to use the embedded hypercube. However, for larger values of p, the increase in the number of hops to communicate between logical neighbours on the embedded hypercube may make it more cost-effective to use the torus itself.

- The nature of the application program - If the phases of the application program comprise only of one recognised function each, or if the selected parallel implementation implements only one recognised function in parallel in a phase, then no sub-division of the topology is required. In such cases, the 2-D torus topology may prove to be more cost-effective than the hypercube embedded in it.

A possible approach would involve computing two sets of costs - one for the 2-D torus itself and another for the hypercube embedded in it - and selecting the more cost-effective implementation for the particular problem.