Chapter 5

The Cost Model

A style of programming is advocated which is based on the set of recognised functions as presented in Chapter 4. An analytical cost model is employed to predict the execution costs for these programs at compile-time. Based on these costs, a cost-effective parallel implementation is selected. The analyser requires some information about machine-dependent parameters, input data sizes and sequential functions, which is discussed in more detail in the following section.

The scheme for obtaining a parallel implementation for a program is shown in Figure 3.1. Chapter 3 gives an outline of the analysis scheme. This chapter discusses the analyser in greater detail.

5.1 The Problem Specification

In principle, the analyser should abstract away from low-level details to the extent possible. However, the costs computed by the analyser should reflect the practical costs accurately. The input to the analyser is in the form of the following tuple:

$$\text{program} = (P, M, D, I_s, F_t, S, C_f, F_s).$$

Symbol	*Meaning*
P	Program Tree
M	Parallel Machine Characteristics
D	Input List Nesting
I_s	Input List Size
F_t	Input List Type
S	Set of Relationships
C_f	Cost of Sequential Function
F_s	Output Type of Sequential Function

The meaning of each symbol will be discussed in the following subsections. Much of the information required could be obtained with minimum programmer interference if profiling [ADM87, Bus93, San93, Sar91] and type-checking capabilities were built into the analyser. However, the current prototype implementation of the analyser does not include these capabilities and the programmer is required to supply the necessary information.

In all future discussions, it is assumed that:

$$\mathcal{H} = \text{the set of recognised functions,}$$
$$\mathcal{F} = \text{the set of sequential functions.}$$

5.1.1 The Program Tree (P)

P is the program tree representing the program. The program tree is indirectly supplied by the programmer; the analyser constructs it from the specification of the program code. Each node in P corresponds to either a recognised function or a user-defined one. Each phase in the program is represented by a branch in the tree. A function G, which is an argument of another function F, is represented as a child node of that representing F, on the branch corresponding to that phase. Since the program is typically a **composition** of phases, each phase is an argument of the **composition** function. The branch corresponding to each phase is therefore represented as a child of the **composition** node. The input lists are not represented on the program tree. However, the number of input lists required

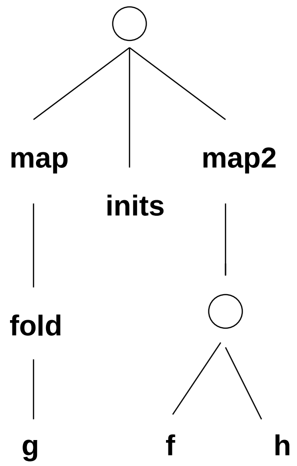

Figure 5.1: An Example of a Program Tree

by each phase is determined by the nature of the recognised functions present in the phase. Henceforth, the terms *branch* and *phase* will be used interchangeably. Therefore, the program,

$$\textbf{map}\ (\textbf{fold}\ \mathrm{g}) \circ \textbf{inits} \circ \textbf{map2}\ (\mathrm{f} \circ \mathrm{h})$$

would have a program tree as shown in Figure 5.1.

5.1.2 The Parallel Machine Characteristics (M)

M is a 4-tuple which describes the characteristics of the parallel machine on which the program is to be executed and has to be supplied by the programmer. The parallel machine is assumed to have p identical processors, each with the same amount of local memory.

$M : string \times Z_0^+ \times Q_0^+ \times Q_0^+$
$M =$ (topology, number-of-processors, K_0, K_1).

Although the program itself is independent of the architecture on which it is executed, the implementation that will be selected would depend on it. The cost model has therefore been *parameterised* on the characteristics of the parallel machine.

- topology - The choice of parallel implementations for the recognised functions is dependent on the interconnection topologies between the processors. Cost models have been studied for some well-known interconnection topologies, such as the *hypercube*, *2-D torus* and *tree*.

- number-of-processors - This specifies the maximum number of processors available on the parallel machine.

- K_0 and K_1 - In a message-passing system, data communication involves two costs - the start-up cost to initiate the communication and the actual cost of transferring the data. The start-up cost, K_0, is usually a significant portion of the communication cost (and can dominate it, especially when the data size is small), and cannot be ignored. The cost for the actual communication of data will depend on the bandwidth of the communication link, K_1, between the two processors. K_0 is expressed in some time unit (e.g. ms) and K_1 is expressed in bytes/time unit (e.g. bytes/ms). Both of these parameters are specific to a given architecture and can be obtained from the machine manufacturer. A linear model is used to compute the communication costs [SS89] between nearest neighbours. Therefore, the cost of communicating n bytes of data between neighbours is given by the following expression:

$$T_c = K_0 + \frac{1}{K_1}n. \tag{5.1}$$

  It is assumed that only data is being communicated, and that each processor possesses a copy of the source code. It is also assumed that processors cannot communicate with more than one neighbour at a time. Therefore, initiating communications with m different neighbours would incur a start-up cost of mK_0 (as opposed to K_0). The algorithms and costs for data communication routines are discussed in Chapter 6.

5.1.3 The Input List Characteristics (D, I_s, F_t, S)

- D represents the level of nesting in each of the input lists.

  D : string $\times Z_0^+$

  string is the variable that represents the list. For example, for a list called *matrix_A* which is a list of lists, the D would be specified as ("matrix_A", 2).

- I_s is a D-tuple which represents the list sizes at levels, $0, 1, \ldots, (D-1)$.

  I_s : string $\rightarrow (\underbrace{Z_0^+ \times \ldots \times Z_0^+}_{D-tuple})$

  For example, for the list *matrix_A* which is a list of lists of size (32×64), I_s would be specified by ("matrix_A", (32,64)). I_s could be estimated by profiling. In the absence of a profiler in the prototype implementation, if the input list size is not known at compile-time, I_s is computed after the specification for S is obtained. It is important to note that the size(s) of the input list(s) is(are) required to be specified only at the beginning of the first phase in the program. For subsequent phases, the analyser deduces the size of the input list, based on the transformations applied to it by the functions in the previous phase. However, for a recognised function with two argument lists, if only one input list arrives from the previous phase, then an estimate of the size of the other list will be required.

- F_t is a function that computes the size of each element in level $(D-1)$ of the input list(s).

  F_t : string $\times$ type $\rightarrow Q_0^+$

  In other words, F_t represents the size of the base element of each of the input lists in the program, where *string* is the variable representing the input list.

This is computed from the *type* of each input list, which could be deduced by a type-checker. For example, for the list *matrix_A* which is a list of lists of integers, F_t would be specified by ("matrix_A", int). The *sizeof(int)* function then computes the required size. Programs in functional languages can be written at high levels of abstraction, where a single program could apply to a whole group of list types. The analyser will, however, force the programmer to specify the type of the base element of each input list. In that sense, some of the power of *abstraction* in functional languages is lost. This information, however, is necessary for the analyser to predict communication costs. In the absence of type information, the data size is not known, making the true communication costs difficult to predict.

- S is a $(\begin{pmatrix} D \\ 2 \end{pmatrix} + D)$ -tuple, expressing the relationship between sizes in different levels (i.e. $0, 1, \ldots, (D-1)$) of the input list(s) and the number of processors.

  $S = (R_0, R_1, \ldots, R_{D-1})$

  $\quad R_i \in [\ll, <, \approx, >, \gg],\ 0 \leq i < D$

  where,

  $n \ll k \Rightarrow 0 < \frac{n}{k} < 0.1$

  $n < k \Rightarrow 0.1 < \frac{n}{k} < 1$

  $n \approx k \Rightarrow \frac{n}{k} \approx 1$

  $n > k \Rightarrow 1 < \frac{n}{k} < 10$

  $n \gg k \Rightarrow \frac{n}{k} \geq 10$

  The set of relationships in S, represents a simple method for expressing constraints on the size of the inputs. The choice of 10 as the factor for expressing the estimates is based purely on pragmatic considerations. For compile-time analysis, some knowledge of the *shape* of the input data is required to guide the selection of a parallel implementation. This might even be different for the same program for different input list sizes. In the absence of a profiler, the specifications in S provide some estimates for the analyser regarding the relative sizes of the input list and the number of available processors. In the presence of profiling information, the specification for S would not be required.

Since the number of processors is known (from the specification in M), and the specifications in S estimate size(s) of the input list(s) in relation to the number of processors, I_s can now be estimated for cost calculations.

5.1.4 Specifications for Sequential Functions (C_f, F_s)

- Costs of Sequential Functions.

  C_f is the cost function for sequential functions in the program.

  $$C_f : G_c \cup G_f$$
  $$G_c : \mathcal{F} \rightarrow Q_0^+$$
  $$G_f : \mathcal{F} \rightarrow (G \rightarrow Q_0^+)$$

  G_f represents the case where the cost of a sequential function is proportional to the size(s) of the input argument(s) and G_c represents the case where the cost is a constant. The cost is expressed in some decided unit, e.g. ms. The cost of a recognised function is a function of the cost of its argument function(s) which could ultimately be a sequential function whose cost is not predetermined. It is therefore necessary to determine the cost of the sequential function(s) in the argument, in order to compute the cost of the recognised function. The costs of sequential functions could be estimated by profiling, in which case the specification for C_f would not be required of the programmer.

An assumption that is automatically made by the analyser is that the sequential functions take the same time to operate on different input data. This assumption arises as a result of the *regularity* restriction that is imposed on problems which can be handled by the scheme (See Section 3.3). For example, in **map** f xs, the cost of f is assumed to be the same for every element of the list xs. This automatically means that sequential functions in this scheme, should not arbitrarily alter the size of the input data. Consider the following example.

```
fun f 0 = [ ]
fun f i = i :: f (i-1)
```

This would produce a list whose size depends on the value of i. However, in **map** f xs, f no longer takes the same amount of time to operate on all the elements of the list xs. The cost of f now depends on the actual value of each list element. This violates the assumption of *regularity* in the context of this scheme. In such cases, the specification of C_f would be difficult and one based on the worst or average-case cost would produce predictions that cannot always be guaranteed to reflect practical costs. Profiling would probably remove this restriction to the extent of allowing sequential functions with costs that depend on the actual input data. However, problems would still be required to have a predictable (*regular*) communication structure in order to be expressible in terms of the recognised functions.

- Outputs from Sequential Functions

  F_s computes the size of the output which is produced by the sequential functions in the program, from the specification of the type of the output of a function. This could, again, be deduced by a type-checker.

$F_s : \mathcal{F} \times \text{type} \rightarrow Q_0^+$

Again, F_s imposes some limitation on the full abstraction power of functional languages. However, this information is also important to the analyser for predicting communication costs. In each phase of the program, the input list is transformed by the functions in that phase. Although the nature of the transformation of the input list(s) by a recognised function is predetermined, the manner in which a sequential function would transform its argument list(s) cannot be determined, if the program is written at a high level of abstraction. A sequential function could transform a base element of one *type* into a another *type*. This would result in a change in the size of the input list for the subsequent phase and the analyser must account for this change. The specification of F_s is a means of providing the analyser with the information necessary to account for such changes in the size of the input list between subsequent phases.

Three types of transformation that can be applied on input lists by sequential functions are of interest to the cost analysis. Not all of them can be accounted for by the analyser in its present form.

1. The output list could be transformed to just the extent that the size of the base element is different from that in the input list. e.g. a list of integers into a list of reals. In this case, the sequential function does not alter the D or I_s specifications and these can be used as such to estimate the costs for the subsequent phase. The only change is in the specification of F_t and this is determined simply by the specification of F_s for the sequential function. If $F_{s'}$ specifies the output size for the sequential function in the phase, then $F_t = F_{s'}$ for the list input to the next phase.

2. The sequential function may transform the list in a manner that alters its D-value, e.g. a list of lists of integers could be transformed into a list of integers. In that case, $D = 2$ for the input list to the sequential function and $D = 1$ for the output list. An example of such a function is *update*, as defined in the Jacobi Iteration example in Section 4.3.3. For the subsequent phase, the analyser must account for the change in the D-value of the input list, if it is to make realistic cost predictions. Such transformations can be deduced by a compiler, but the analyser does not incorporate such facilities at present, and is left for future work.

3. The sequential function could transform the input list so as to alter its I_s specification, e.g. a list of lists of size $(m \times n)$ could be altered to a list of lists of size $(m \times k)$. An example of such a function is the *rearrange* function as defined in Section 4.3.3. It transforms an input list of size $(3 \times n)$ into one of size $(n \times 5)$. Such transformations cannot, in general, be deduced at compile-time. In such cases, an inaccurate size of the input list will be deduced for subsequent phases. This in turn might lead to poor cost predictions and in the worst case may lead to the selection of an inefficient parallel implementation for the program. This is a disadvantage of the scheme, at present. Again, a profiler could rectify the situation to some extent.

Sequential functions could cause transformations on input lists, so that some combinations of 1, 2, 3 are produced. The arguments remain the same.

5.2 The Compile-time Analysis

In the HOPP model, an application program is a **composition** of phases. The **composition** itself is performed sequentially, i.e. phase i does not commence until phase $(i-1)$ is completed. This need not be the case and future work could consider pipelining the phases. The results from phase $(i-1)$ can then be input to phase i as they become available. In the absence of pipelining, the total cost of the program comprising of k phases is given by:

$$C_p = \sum_{i=1}^{k} C_{p_i} + \sum_{i=0}^{k-1} C_{i,i+1}$$

where, C_{p_i} represents the cost of phase i, and $C_{i,i+1}$ represents the cost of phase transition between phases i and $i+1$, respectively. The cost of a phase depends on the nature and cost of its composing functions. The cost of phase transition represents the communication cost incurred in rearranging the output data of one phase to be suitable for the subsequent one. Costs for communication routines such as *scatter*, *gather*, *total exchange* and *broadcast*, on the different processor topologies have been derived (see Chapter 6 for details). The rearrangement of data between consecutive phases would typically involve one or more of these communication operations. The analyser uses the costs of these routines in computing the costs of phase transition. If no data movement is required between two consecutive phases, then the cost of phase transition for those two phases is zero.

The following points should be noted.

1. Each recognised function operates on a list data structure.

2. Every recognised function has an associated parallel implementation on each target machine topology included in the model.

3. The cost of implementing a recognised function, F, in parallel on p processors, on an input list of size n is represented by:

$$C = F(n, p, C_a)$$

where, C_a, is the cost of the argument function (if any). In general, if the recognised function, F, operates on k input lists of sizes $n_1, n_2, \ldots, n_k$ respectively, then the cost is represented by:

$$C = F((n_1, n_2, \ldots, n_k), p, C_a).$$

The analyser performs a cost analysis for each branch in the program tree. A branch in a program tree might comprise of instances of both recognised and sequential functions. If a branch (phase) comprises entirely of sequential functions, then the analyser would be forced to select a sequential implementation for it.

Let p be the number of processors and C_s be the cost of the sequential argument function (if any). In the following sections, the representation of cost expressions assume that the recognised functions operate on a single input list.

- The branch could contain only one occurrence of a recognised function, F, in which case only one parallel implementation is possible for a particular topology. If the input list is assumed to be of size m, then the cost of the branch is given by:

$$C_1 = F(m, p, C_s). \tag{5.2}$$

- The branch could contain a recognised function, F, that has another recognised function, G, as its argument. The input data structure would be a list of lists ($D \geq 2$). If the data structure is assumed to comprise of m lists, each of which contains n elements, then three different parallel implementations are considered.

1. One in which the function F is implemented in parallel on p processors, and the function G sequentially. The cost in this case is given by:

$$C_2^1 = F(m, p, G(n, 1, C_s)). \tag{5.3}$$

2. One in which the function F is implemented sequentially, and the function G in parallel on the p processors. The cost in this case is given by:

$$C_2^2 = F(m, 1, G(n, p, C_s)). \tag{5.4}$$

3. The case where both the functions, F and G, are implemented in parallel. The p-processor machine is divided into p_1 parts, each of which comprises of p_2 processors, where, $p_1 \times p_2 \leq p$. The function F is implemented in parallel on p_1 processors, and the function G is implemented in parallel on p_2 processors. Each of these parts retains the same logical topology as the original machine. However, on certain topologies, logical neighbours may no longer be physical neighbours after such a division. In such a case, it would take more than one hop to communicate between logically neighbouring processors, thereby increasing the communication cost. In order that these increased communication costs are accounted for, the cost function is parametrised on the maximum number of hops to be traversed between logical neighbours, and is represented by l. The value of l would depend on the nature of the topology. The cost in this case is given by:

$$C_2^3 = F(m, (p_1, l_1), G(n, (p_2, l_2), C_s)). \tag{5.5}$$

   If $l_1 = l_2 = 1$, Equation 5.5 can be reduced to:

$$C_2^3 = F(m, p_1, G(n, p_2, C_s)). \tag{5.6}$$

As an illustration of such an implementation, a p-processor hypercube of dimension, $d = \log_2 p$, is divided into 2^k $(0 < k < d)$ smaller hypercubes (subcubes). Each subcube comprises of 2^{d-k} processors. The m lists are scattered across the 2^k subcubes, $\lceil \frac{m}{2^k} \rceil$ per subcube. The $\lceil \frac{m}{2^k} \rceil$ elements all go to a single processor in each of the subcubes, namely, to processors numbered by

$$0, 2^{d-k}, 2.2^{d-k}, 3.2^{d-k}, \ldots, (2^k - 1).2^{d-k}.$$

This implies a parallel implementation for F on 2^k processors, with $\lceil \frac{m}{2^k} \rceil$ elements per processor. Then the function G can be evaluated in parallel on 2^{d-k} processors, with $\lceil \frac{n}{2^{d-k}} \rceil$ elements per processor. Since all the subcubes are connected, and each subcube retains the same connectivity as the original hypercube, $l_1 = l_2 = 1$. The cost in this case is given by:

$$C_2^h = F(m, 2^k, G(n, 2^{d-k}, C_s)). \tag{5.7}$$

In all the three cases, the cost of an inner-level function is passed as a parameter to the outer-level function. At the innermost level this might correspond to a sequential function for which no known parallel solution exists. Information regarding the number of processors available and the input data sizes, is passed down from the outer function to the inner function. This information would depend on the implementation that is selected. The inner function then evaluates its own cost, based on the information it receives. Information about the cost of the inner function is then passed back up to the outer function, which can then evaluate its cost. This argument could be applied to any number of levels in a branch in the program tree.

- The branch could contain three or more occurrences of recognised functions. First, the case when the branch contains exactly three recognised functions, F, G and H, is considered. G is the argument of F, and H is the argument of G. The input data structure is a list of list of lists ($D \geq 3$). Seven possibilities arise, corresponding to the implementations of each of the three functions in parallel, any two in parallel and all three in parallel. For the first six cases, arguments similar to the previous case can be applied and the costs can be similarly derived. If the input data structure is assumed to comprise of m lists, each of which comprises of n lists, each of which in turn contains k elements, then the cost of the branch corresponding to implementing all three functions in parallel is given by:

  $$C_3 = F(m, (p_1, l_1), G(n, (p_2, l_2), H(k, (p_3, l_3), Cs))) \qquad (5.8)$$

  where, $p_1 \times p_2 \times p_3 \leq p$, and l_1, l_2, l_3 are defined as before.

  Theoretically, the occurrence of more than three recognised functions on a branch could be considered. This would mean a data structure that is a list of lists of lists ($D \geq 4$). If such a case does arise, the HOPP model exploits potential parallelism only up to the first three levels and the discussion in the previous paragraph applies.

It is clear that for every phase in the program, there may be several possible parallel implementations, depending on the number of recognised functions in that phase. Under the current scheme, the maximum number of implementations that can be considered for a phase is *eight* (including a sequential implementation). If all the possibilities are considered at every phase, the analysis results in the construction of a search tree. The weights on the nodes of the search tree at each level represent the costs of the different possible implementations for each phase. This cost includes computation as well as any communication costs that may be incurred in the implementation of the recognised function(s) in that phase. The weights on the edges of the search tree represent the costs of phase transition and comprise only of communication costs. A traversal of each path in the tree yields a cost of implementation for the whole program. Each such cost represents the cost of implementing the program when some particular sequence of implementations is chosen for the phases in the program. Each path in the search tree represents a unique and complete implementation for the entire program. The least-cost path in the search tree corresponds to the most cost-effective implementation for the program. The analyser first performs a cost analysis on the program and constructs the search tree at compile-time. The search tree also contains information about the functions in each phase, the number of processors to be used for the implementation of each recognised function in the phase, the nature of the data distribution at the end of each phase, and so on. Using the information on the branches in the path that is selected, code for the implementation can be generated and a copy distributed to all the processors. This would contain code for the recognised and sequential functions in each branch as well as for performing any communications required in the implementation of the program.

It is clear that the size of the search tree grows exponentially with the number of phases in the program. It is difficult to provide a threshold value for the number of phases in the program, after which the search space could be termed as *unacceptably large*. However, even for the Jacobi Iteration example comprising of seven phases (see Section 4.3.3), the search space was quite large and it would have been helpful to reduce it. Heuristics to prune the exponential growth of the search tree have to be investigated. Rather than considering all possible implementations, those that are likely to result in high execution costs could be identified and discarded at the analysis stage itself.

It is important to note that low-level costs such as memory access times are not accounted for. This would make the model very machine-specific and is not desirable. List processing costs, which include the costs incurred in constructing or traversing a list, are however, estimated by the analyser since these costs can prove to be significant. The analyser estimates these costs based on the nature of the input list (obtained from I_s and F_t) and the details are transparent to the programmer. During the analysis of the program, the analyser actually performs list construction or traversal operations on a dummy list that is of the same type as the specified input list and estimates the cost, which is then used as the list processing cost. Since all the low-level costs associated with a practical implementation are not accounted for by the analyser, the actual cost of implementation is expected to be greater than that predicted. However, computing the *absolute* cost is not necessary, since the strategy is to select an implementation based on cost *comparison*. There are four different costs involved.

- The theoretically predicted cost of sequential execution for the program, represented by T^s_{theor}.
- The theoretically predicted cost of the selected parallel implementation on p processors, represented by T^p_{theor}.
- The actual cost of sequential execution for the program when executed on a single processor on the parallel machine, represented by T^s_{prac}.
- The actual cost of the selected implementation for the program when executed on p processors on the parallel machine, represented by T^p_{prac}.

If the model reflects the practical performance, then

$$\frac{T^p_{theor}}{T^s_{theor}} \approx \frac{T^p_{prac}}{T^s_{prac}}. \tag{5.9}$$

5.2.1 The Algorithm for the Analyser

Algorithm **ANALYSE** presents a high-level description of the analyser. The function GET_SPEC obtains the problem specification as discussed in Section 5.1. The function, CONSTRUCT_PROGRAM_TREE, constructs the program tree from the specification of the program code. If a branch in the program tree contains only one recognised function, then in the following algorithm the input list is represented by n_1 and the cost of the recognised function is represented by R_1. If the branch contains two recognised functions, then the input list is represented by (n_1, n_2) and the costs of recognised functions are represented by R_1 and R_2, respectively. For a branch containing three recognised functions, the input list is represented by (n_1, n_2, n_3) and the costs of the recognised functions are represented by R_1, R_2 and R_3, respectively. C_s represents the cost of a sequential argument function. Phase$_i$ of the program is assumed to be implemented on p_i processors; $p_0 = 1$, implying that the data is initially resident on a single node of the parallel machine. N_i represents the number of recognised functions in a phase, $0 \leq N_i \leq 3$. The following rule is used in the derivation of cost expressions:

$$\text{If } k = N_i + 1, \text{ then } R_k(n_k, p, C_a) \equiv C_s^k$$

It means that R_k is not a recognised function and its cost must, therefore, be replaced by the cost of the sequential function at the k^{th} level.

The function ADD_TO_S_TREE inserts a child node to a specified node in the search tree. In the implementation, the weight on the inserted branch is added to the weight on the destination node. In other words, the inter-phase transition cost is added to the cost of implementation of the subsequent phase. C^i_{seq} represents the cost of implementing phase$_i$ sequentially. $C^i_{par\_x}$ represents the cost of implementing function R_x in phase i in parallel $(1 \leq x \leq 3)$, $C^i_{par\_xy}$ represents the cost of implementing both the functions R_x and R_y in phase i in parallel on p_i^x and p_i^y processors respectively $(1 \leq x \leq 2, 2 \leq y \leq 3)$, $(p_i^x p_i^y \leq p_i)$ and $C^i_{par\_123}$ represents the cost of implementing functions R_1, R_2 and R_3 in phase i in parallel on p_i^1, p_i^2 and p_i^3 processors respectively $(p_i^1 p_i^2 p_i^3 \leq p)$.

The function REARR computes the cost for rearranging the output data of one phase to suit the implementation of the subsequent phase. The information about the current data distribution is obtained from the current parent node in the search tree. If the output list of size n_1 from phase $i-1$ is distributed across p_{i-1} processors, and phase i requires the data list to be distributed across p_i processors, $(p_{i-1}, p_i \leq p)$, then REARR (n_1, p_{i-1}, p_i) computes the required data rearrangement cost. Similarly, for the list (n_1, n_2) to be distributed such that the sublists are distributed across p_i^1 processors and each sublist across p_i^2 processors $(p_i^1 p_i^2 \leq p)$, REARR $((n_1, n_2), p_{i-1}, (p_i^1, p_i^2))$ computes the required data rearrangement cost.

S_tree represents the search tree, and *S_lev* represents the current level in the search tree. Initially, *S_tree* simply comprises of a root node which is at level zero and has zero cost. The analysis of each branch in the program tree could insert up to eight child nodes (corresponding to the maximum number of implementations considered by the analyser), to each node of the search tree at the current level. In the algorithm, the inter-phase transition cost is added to that of each such implementation in order to compute its total cost.

The function ADD_TO_S_TREE inserts a child node to a specified node in the search tree. In the implementation, the weight on the inserted branch is added to the weight on the destination node. In other words, the inter-phase transition cost is added to the cost of implementation of the subsequent phase. C^i_{seq} represents the cost of implementing phase$_i$ sequentially. $C^i_{par\_x}$ represents the cost of implementing function R_x in phase i in parallel $(1 \leq x \leq 3)$, $C^i_{par\_xy}$ represents the cost of implementing both the functions R_x and R_y in phase i in parallel on p_i^x and p_i^y processors respectively $(1 \leq x \leq 2, 2 \leq y \leq 3)$, $(p_i^x p_i^y \leq p_i)$ and $C^i_{par\_123}$ represents the cost of implementing functions R_1, R_2 and R_3 in phase i in parallel on p_i^1, p_i^2 and p_i^3 processors respectively $(p_i^1 p_i^2 p_i^3 \leq p)$.

The function REARR computes the cost for rearranging the output data of one phase to suit the implementation of the subsequent one. The information about the current data distribution is obtained from the current parent node in the search tree. If the output list of size n_1 from phase $i-1$ is distributed across p_{i-1} processors, and phase i requires the data list to be distributed across p_i processors, $(p_{i-1}, p_i \leq p)$, then REARR (n_1, p_{i-1}, p_i) computes the required data rearrangement cost. Similarly, for the list (n_1, n_2) to be distributed such that the sublists are distributed across p_i^1 processors and each sublist across p_i^2 processors $(p_i^1 p_i^2 \leq p)$, REARR $((n_1, n_2), p_{i-1}, (p_i^1, p_i^2))$ computes the required cost.

S_tree represents the search tree, and *S_lev* represents the current level in the search tree. Initially, *S_tree* simply comprises of a root node which is at level zero and has zero cost. The analysis of each branch in the program tree could insert up to eight child nodes (corresponding to the maximum number of implementations considered by the analyser), to each node of the search tree at the current level. In the algorithm, the inter-phase transition cost is added to that of each such implementation in order to compute its total cost.

```
Algorithm ANALYSE
BEGIN
  GET_SPEC          /* obtain problem specification */
  CONSTRUCT_PROGRAM_TREE (P)
  S_tree ⟵ root;   S_lev ⟵ 0       /* initialise search tree */
  FOR i ⟵ 1 to number-of-branches in P DO
    N_i ⟵ number of recognised functions in branch_i    /* ≤ 3 */
    j ⟵ 1
    FOR each node_j in level S_lev of S_tree DO
      Compute C^i_seq      /* sequential implementation */
      ADD_TO_S_TREE (S_tree, S_lev, j, C^i_seq)
      FOR x ⟵ 1 to N_i DO
        FOR k ⟵ 1 to N_i DO
          IF (k = x) THEN p_k ⟵ p_i ELSE p_k ⟵ 1
        C^i_par_x ⟵ REARR (n_x, p_{i-1}, p_i) +
                  R_1(n_1, p_1, R_2(n_2, p_2, R_3(n_3, p_3, C_s)))
        ADD_TO_S_TREE (S_tree, S_lev, j, C^i_par_x)
      FOR x ⟵ 1 to N_i - 1 DO
        FOR y ⟵ 2 to N_i DO
          IF (x ≠ y) THEN
            FOR k ⟵ 1 to N_i DO
              IF (k = x) THEN p^k_i ⟵ p^x_i
              ELSE IF (k = y) THEN p^k_i ⟵ p^y_i ELSE p^k_i ⟵ 1
            C^i_par_xy ⟵ REARR ((n_x, n_y), p_{i-1}, (p^x_i, p^y_i)) +
                R_1(n_1, (p^1_i, l_1), R_2(n_2, (p^2_i, l_2), R_3(n_3, (p^3_i, l_3), C_s)))
            ADD_TO_S_TREE (S_tree, S_lev, j, C^i_par_xy)
      IF (N_i = 3) THEN
        C^i_par_123 ⟵ REARR ((n_1, n_2, n_3), p_{i-1}, (p^1_i, p^2_i, p^3_i)) +
                R_1(n_1, (p^1_i, l_1), R_2(n_2, (p^2_i, l_2), R_3(n_3, (p^3_i, l_3), C_s)))
        ADD_TO_S_TREE (S_tree, S_lev, j, C^i_par_123)
    S_lev ⟵ S_lev + 1;
END.
```

The algorithm in the actual implementation also checks that the number of processors available is sufficient to implement more than one recognised function in parallel, before further analysis is carried out. If there are insufficient processors for any implementation, then the corresponding branches will not be added to the search tree. Also, if two recognised functions are to be implemented in parallel, then the most cost-effective partition of the processor network is first determined and the costs corresponding to this partition are inserted in the search tree. For the sake of brevity, these and a few other low-level details of the analysis are not shown in the algorithm.

5.3 An Example of Compile-time Analysis

To provide a flavour for the manner in which the analyser handles the input specifications, a simple example is considered.

fun ex xs = (**s_fold** mult $\circ$ **map** (**s_fold** plus)) xs;

The recognised functions are represented in boldface. The functions *mult* and *plus* are defined to be functions that multiply and add two integers respectively.

The program comprises of two phases. The first phase contains two recognised functions, **map** and **s_fold**, and the second phase contains one recognised function, **s_fold**. The analyser first requires the problem specification, as discussed in Section 5.1. In this simple example, the input sizes are assumed to be known at compile-time.

The program tree, P, representing the program is depicted by Figure 5.2.

The parallel machine is assumed to comprise of p processors connected as a hypercube. The dimension of the hypercube is represented by d. K_0 and K_1 represent the communication start-up cost and communication channel bandwidth on the hypercube, respectively. The specification for M for both the phases is:

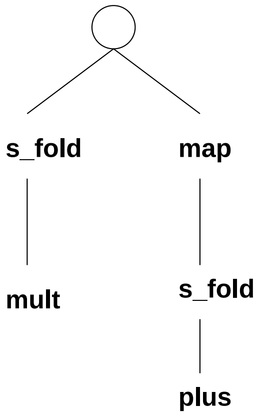

Figure 5.2: Program Tree for the Example

$M \equiv (p$, "hypercube", K_0, $K_1)$.

Since the input data sizes are assumed to be known at the start of the program and the analyser deduces it for the second phase, $S \equiv \emptyset$, for both the phases.

First, the specifications for *phase one* are considered.

- $C_f \equiv$ Cost of *plus*, represented by C_p.
- $D \equiv 2$, since the input data structure is a list of lists, in which each sublist is a list of integers. (This could be deduced from the type of the function *plus*).
- I_{sa} is specified since it is assumed that the input data sizes are known at compile-time. Let the size of the input list be represented by $(m \times n)$ - i.e. the list comprises of m sublists, each of which contains n integers.
- $F_t \equiv$ (size of integer).
- $F_s \equiv$ (size of integer), since the function *plus* adds two integers and outputs an integer.

In this example, D, F_t and F_s could be deduced from the *type* of the function *plus*, if the analyser incorporated a type-checker. C_f and I_{sa} could be estimated if the analyser incorporated a profiler.

The input size specifications are made available (either by the programmer or by profiling techniques) at the beginning of the first phase. The analyser computes the data sizes for the subsequent phases from the nature of the recognised functions which are currently being analysed. In the example program, the input to the first phase is a list of lists of size $(m \times n)$. The **s_fold** in *phase one* reduces this to a list of size m, in which each element is an integer. The analyser deduces the following attributes for *phase two*.

- $D \equiv 1$, since a list of lists of integers has been reduced to a list of integers.
- $I_{sa} \equiv m$.

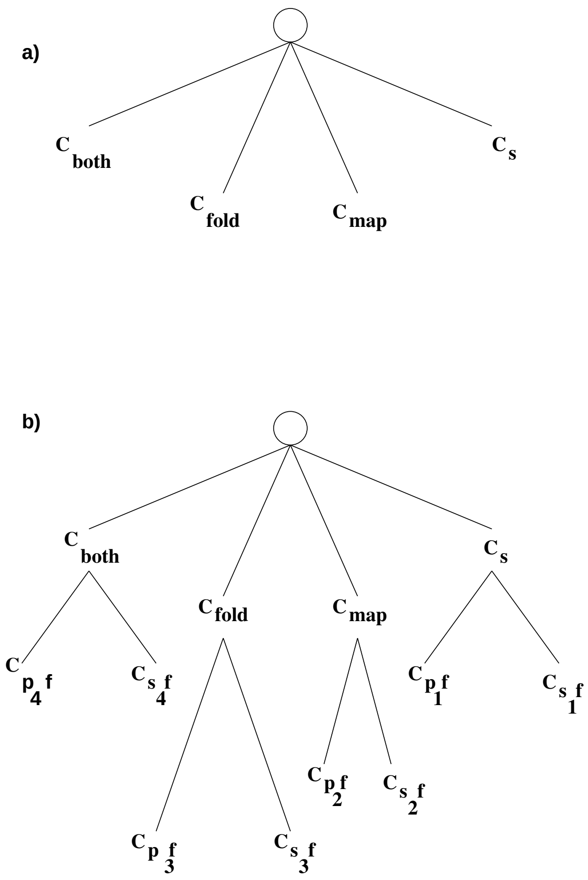

Figure 5.3: Search Tree for the Example

A type-checker and a profiler (or the programmer) provide the following information:

- $C_f \equiv$ Cost of *mult*, represented by C_m.
- $F_t \equiv$ (size of integer). This again, could be deduced from the *type* of the function *mult*.
- $F_s \equiv$ (size of integer), also deducible from the *type* of the function *mult*.

Both functions can be implemented in parallel only if the dimension, d, of the hypercube ≥ 2. Figure 5.3a represents the search tree after *phase one*. (Since **composition** works right to left, the rightmost branch on the program tree is executed first). C_s represents the cost of sequential implementation for the phase, C_{map} represents the cost of implementing **map** in parallel and **s_fold** sequentially, C_{fold} represents the cost of implementing **s_fold** in parallel and **map** sequentially and C_{both} represents the cost of implementing both **map** and **s_fold** in parallel.

Figure 5.3b represents the search tree after *phase two*. In *phase two*, the analyser computes the costs for two possible implementations for **s_fold** - the sequential and parallel implementation, respectively, and the corresponding costs are represented by $C_{s_i f}$ and $C_{p_i f}$, $1 \leq i \leq 4$. For each branch in *level two* of the search tree, the costs of implementation include the costs incurred in re-distributing the data from *phase one* to *phase two*. For some branches, (e.g. $C_{p_2 f}$), no data re-distribution is required.