# Chapter 3

# Overview of the HOPP Model

## 3.1 Introduction

This chapter presents an outline of the HOPP approach to parallel programming. HOPP is based on an implicitly parallel language whose constructs are borrowed from the Bird-Meertens Formalism (BMF) [Bir89] and FP [Bac78]. These constructs are essentially higher-order functions which perform useful operations on lists. Most of these functions are inherently parallel and will henceforth be referred to as *recognised functions*. Since the behaviour of the recognised functions is predetermined, a program which is expressed in terms of these functions can be analysed at compile-time to realise a cost-effective parallel implementation. However, such an analysis is only possible for *regular* problems which are expressed in terms of the recognised functions. The implications of *regularity* in this context are discussed in Section 3.3. The programs are targeted at distributed-memory machines in which the communication costs tend to be significant.

## 3.2 The Features of the Model

The HOPP model comprises of three parts - the *program* model, the *machine* model and the *cost* model.

- The program model - A program in HOPP is a *composition* of nested instantiations of recognised and user-defined functions. *Composition* in this context refers to functional composition and works from right to left. Each component is referred to as a *phase* of the program.

- The machine model - The programs are targeted at distributed-memory MIMD machines which consist of a set of processors connected by an interconnection network. The topologies of machines considered include the hypercube, 2-D torus, linear array and tree. The data is distributed among the processors of the machine.

- The cost model - For each recognised function, the cost model determines its cost of parallel execution on a given topology. Effectively, the cost model computes the costs of possible implementations of a given program on a given machine topology. The theoretical cost model has been implemented in the form of an *analyser*.

## 3.3 Language Assumptions for the Model

As already mentioned, the HOPP model is based on a functional paradigm. This section describes the structure of a program in HOPP, along with the implications for parallel implementations which will be considered. The meaning of *regularity* in this context is also discussed.

The only data structure is the list, on which all the recognised functions operate. Lists can be arbitrarily nested and of any type, including standard or user-defined. The current prototype implementation of the analyser and parallel code library only allow lists whose base elements are of standard type or pairs of standard types, but this is only a limitation of the implementation. A further assumption which is made by the analyser is that sublists are of equal length.

This is implicit in the assumptions of *regularity*, which will be discussed shortly.

At the top level, a program is a composition of functions. Each component (phase) could correspond to a nested instantiation of recognised and user-defined functions. A program could consist of one or more phases and this is the *only* allowable structure for a program in HOPP. The following is an example of a program in HOPP. *plus* and *times* correspond to user-defined functions, which add and multiply two integers, respectively. The program comprises of three phases, each with instances of recognised (represented in boldface) and user-defined functions.

prog xs = (**s_fold** plus 0 ∘ **map** (**s_scan** plus 0)
∘ **r_cross_product** times ys) xs;

If the topmost level of a phase corresponds to a user-defined function, then any instances of recognised function(s) that it might have as its argument(s) is(are) not considered for parallel implementation. If the topmost level of a phase comprises of a recognised function, then parallel implementations are considered for its argument recognised function. This procedure is followed up to a maximum of three levels or until a user-defined function is encountered. This implies that parallel implementations are considered for nested recognised functions up to three levels and any recognised functions below it are treated as user-defined functions and implemented sequentially. The reasons for considering parallel implementations for up to only three levels are purely pragmatic, and will be explained more clearly in Section 3.5.

Consider the following examples.

prog_1 = **map** (**s_fold** plus 0) xss;

prog_2 = **map** S xss;

fun S xs = **s_fold** plus 0 xs;

```
prog_3 = map (s_fold plus 0 ∘ map sqr) xss;
prog_4 = map T xss;
          fun T xs = let val ys = map sqr xs in
                         s_fold plus 0 ys
                     end;
```

The recognised functions are depicted in boldface and ML notation is used. The function *plus* simply adds two integers and the function *sqr* produces the square of an integer.

All the four programs consist of a single phase with nested instantiations of recognised and user-defined functions, in the restrictive format allowed by the HOPP model. *prog_1* and *prog_2* are equivalent in a sequential setting. However, **s_fold** is encapsulated within the user-defined function, *S*, in *prog_2*. The current implementation of the HOPP model will only *recognise* **map** as a potential function for parallel execution in *prog_2*, and **s_fold** within *S* will only have a sequential implementation. However, in the case of *prog_1*, **s_fold** is at the top level and will, therefore, be *recognised* as a candidate for parallel evaluation, in addition to **map**. Again, *prog_3* and *prog_4* perform the same functions, but the former has more identifiable parallel implementations than the latter. In *prog_4*, a parallel implementation will be considered only for **map**, but none of the instances of recognised functions within *T* would be treated as such. The limitation arising from cases such as *prog_2* can be easily rectified, by checking for instances of recognised functions in the top level of user-defined functions. However, for cases such as *prog_4*, where instances of recognised functions can be arbitrarily nested within user-defined functions, the problem becomes harder to tackle.

User-defined functions are implemented in a chosen strict functional language, following its syntactic rules. The HOPP model is independent of the base language. However, user-defined functions are not allowed to perform input-output operations. The data is input on a single processor which distributes it to other processors as required, during the intermediate phases of the program. The output is produced at the end of the last phase in the program and could be left distributed across the processors. This is the only allowable form of data flow, which could be violated by allowing sequential functions to perform input-output operations.

The implications of *regularity* in this context restrict polymorphism. The analyser needs type information at compile-time in order to compute the size of the base elements in the input list at every stage in the program. An accurate knowledge of this size is crucial to the computation of communication costs. Consequently, definitions of sequential functions that allow for full or restricted polymorphism are not permitted. The type-checker in the analyser would force the specification of the required type information. The following examples illustrate the point.

ex_1 = **map** g xs;

    fun g (x,y) = ((if (y > 0) then ~y else y), x);

ex_2 = **map** all_eq xs;

    fun all_eq (x,y,z) = (x = y) andalso (y = z);

In *ex_1*, the user-defined function $g$ is polymorphic in $x$. Consequently, its size cannot be deduced at compile-time and the analyser will not allow such a definition. The programmer will be forced to specify the *type* of $x$. In *ex_2*, the user-defined function *all_eq* is a valid restricted polymorphic function. However, since the types of $x$, $y$ and $z$ cannot be deduced at compile-time, such a definition cannot be allowed. It may be noted that both of the above definitions will be allowed if type information is explicitly specified.

A regular program in this context is related to what is termed as a *shapely* one in [Jay95]. The analyser performs operations similar to *shape analysis*, whereby, given the *shape* of the inputs, the *shapes* of all the intermediate values and that of the result can be deduced. For a program that is not regular (*shapely*), these deductions will not, in general, be possible, leading to poor performance prediction. Shapely programs also have predictable communication structures. The latter is imposed by the use of recognised functions which only allow the expression of certain types of computation. This means that it would be difficult to express many irregular problems using the set of recognised functions.

Regularity also implies that the performance of a program does not vary drastically for different data sets. This is a limitation arising due to compile-time analysis, but cannot be enforced by the analyser. Although it is possible to write problems which are not regular in this sense, it cannot be guaranteed that the behaviour predicted by the analyser will be obtained experimentally in such cases. (Refer to Section 5.1.4 for an example).

## 3.4 The Advantages and Limitations of HOPP

The motivation behind choosing a language based on BMF as a model of parallel computation lies in the advantages that it offers.

- The HOPP model is based on a functional paradigm. It therefore automatically inherits all the advantages of functional programming as described in Chapter 2.

- BMF incorporates functions that perform operations which are characteristic of several common parallel programming paradigms.

- A program which is expressed in terms of the recognised functions is analysed *statically* to realise a cost-effective parallel implementation. This is possible because the behaviour of the recognised functions is predetermined.

Since all the analysis is performed at compile time, it saves on overheads at run time. However, the analyser would require information regarding machine-specific parameters and *shapes* of input lists.

- The recognised functions are either already part of, or can be easily defined in any existing functional language. HOPP does not impose any new programming technique on the programmer. Simplicity of learning and use are naturally inherited by the model.

There are, however, some limitations to the scheme.

- As discussed in Section 3.3, the analysis can be assumed to reflect experimental behaviour only for *regular* problems.

- For programs that do not contain any occurrences of recognised functions, a parallel implementation cannot be realised. This forces the programmer to remain within the fixed repertoire of available functions.

- The current implementation uses the list as the main data structure and this poses problems relating to efficiency. A sequential implementation incurs overheads due to list creation, destruction and garbage collection. Also, list access is linear in the length of the list as opposed to the constant-time access of the array. A parallel implementation using list data structures suffers from an additional overhead which is incurred during the communication of lists between processors. In order to ensure that pointer references are accurate after communication, the pointer addresses must be converted to offsets at the sending end. The processor that receives the list must then compute the real addresses from the offsets. This effectively increases communication costs. However, this limitation is not a direct consideration of the HOPP model.

## 3.5 The Parallelisation Scheme

A program in HOPP is expressed as a *sequence* of phases. Each phase may contain recognised functions along with instances of user-defined functions. Each recognised function has a predefined parallel implementation on a given target machine topology, along with an associated implementation cost. A knowledge of the target machine topology is essential in order to select a cost-effective implementation associated with that topology for the recognised function. Each implementation attempts to make optimal use of the machine connectivity in an effort to reduce the communication overhead. The performance of the recognised functions on the hypercube, 2-D torus, binary tree and linear array topologies have been studied and will be discussed in Chapter 6. User-defined functions only have a sequential implementation and will henceforth be referred to as *sequential functions*.

In the current scheme, parallelism is only exploited within each phase. The phases themselves are sequential and phase $i$ does not commence until phase $i-1$ is completed. However, future work could consider pipelining as an option, in order to evaluate the phases in parallel. A phase that does not contain any occurrences of recognised functions is implemented sequentially. The parallelisation strategy exploits parallelism in nested recognised functions up to the first three levels. The number of levels exploited for parallelisation is limited to *three* for pragmatic reasons. A phase that has its top three nested functions as recognised ones, will have an input data structure that is at least a list of list of lists and eight possible implementations will be considered for it (see Section 3.5.1). Exploiting more levels for parallelism would further increase the number of implementations to be considered. Parallel implementations are not considered for recognised functions which are nested within sequential functions.

The HOPP model enables architecture-independent parallel programming.

Only *one* program is written, irrespective of the architecture at which it is targeted. However, the decisions which influence the selection of a cost-effective parallel implementation for the program, are dependent on the characteristics of the target architecture. These decisions are now transparent to the programmer. The model allows for portable programs to be written, and at the same time hopes to achieve a realistic reflection of parallel program performance. The model is therefore *parametrised* on the characteristics of the target architecture.

The Bird-Meertens Formalism includes a basic set of higher-order functions. This set has been extended to incorporate a number of *additional* functions. The definitions for all the recognised functions can be found in Chapter 4. It will be shown that all of these functions can be expressed in terms of one or more of the basic set of functions. They have been included as recognised functions in their own right because they are found to be useful in many common problems. The definition, in terms of existing functions, is rather contrived for some of the functions in the extended set. It would therefore save time and effort for the programmer if these functions are already available as recognised functions. It would also make the program itself more readable. More importantly, the scheme attempts to provide a more efficient implementation for programs expressed in terms of these additional functions, in comparison to one based solely on the functions in the basic set.

### 3.5.1 The Analyser

The outline of the parallelisation scheme is depicted in Figure 3.1. The application program is input to the analyser which first constructs a *program tree*. Each branch in the tree corresponds to a phase of the program. A cost analysis is then carried out on the program.

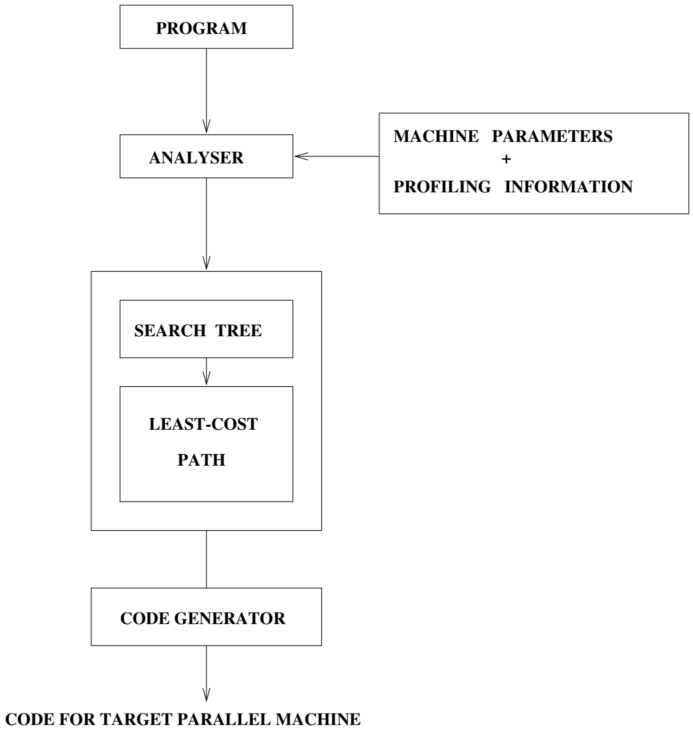


Figure 3.1: The Analysis and Implementation Scheme

The cost of a program comprising of $n$ phases is given by:

$$Cost = \sum_{i=1}^{n} C_{p_i} + \sum_{i=0}^{n-1} C_{i,i+1}$$

where, $C_{p_i}$ is the cost of phase $i$ and $C_{i,i+1}$ represents any communication cost that may be incurred in rearranging the output data of phase $i$ to suit the implementation of phase $i+1$. The cost of a phase depends, among other things, on the nature and the number of recognised functions in that phase and also the parallel implementation selected for that phase, on a given $p$-processor network. In the present scheme, a phase that has only one occurrence of a recognised function has only one parallel implementation, namely, the parallel implementation for that function on the particular network. For a phase containing two recognised functions, one of which is the argument of the other, three parallel implementations are possible.

- A parallel implementation for the outer function on $p$ processors.
- A parallel implementation for the inner function on $p$ processors.
- A parallel implementation for both functions.

The first two implementations are straightforward. The manner in which the third implementation is handled would depend on the type of the underlying processor network. For example, in the case of a $d$-dimensional hypercube, the hypercube is divided into $2^k$ $(0 < k < d)$ smaller hypercubes, with each smaller hypercube containing $2^{d-k}$ processors. The outer function is evaluated in parallel across $2^k$ hypercubes, with the inner function being evaluated in parallel across $2^{d-k}$ processors.

A similar argument can be extended to a phase containing three or more recognised functions, leading to a total of seven possible parallel implementations - three possibilities arising from implementing only one recognised function in parallel, three possibilities arising from implementing any two functions in parallel and the case in which all the three functions are evaluated in parallel. However, parallel implementations are only considered for up to three recognised functions in a phase. Any recognised function(s) below the third recognised function are implemented sequentially. A phase can therefore have at most eight possible implementations including a sequential one. It is important to consider the sequential implementation as well, since it may prove to be the least-cost implementation in some of the cases.

### 3.5.2 The Search Tree

The costs associated with all possible implementations for each of the phases are estimated by the analyser and a search tree is constructed. The nodes at level $i$ of the tree correspond to the costs associated with the different implementations for that particular phase. These costs include both computation costs and communication costs. The most efficient implementation for the whole program is determined by the least-cost path in the search tree, for which code can then be generated and executed on the parallel machine.

It is important to realise that the analyser does not account for costs arising from low level operations such as memory accesses. This would make the analyser very machine-specific. The selection of the least-cost implementation depends only on cost comparisons and therefore the absolute costs are not crucial. List processing costs are, however, accounted for by the model since this forms a substantial overhead in functional languages. This includes the costs which are incurred in constructing a new list or traversing a list. The analyser estimates these costs based on the nature of the input list and the details are transparent to the user.

It is clear that the size of the search tree will grow exponentially with the number of phases in the program. In this thesis, only programs that comprise of a few phases are discussed, and this keeps the search tractable. However, for problems comprising of a large number of phases, some heuristics for pruning the search tree would have to be considered. This issue is discussed in greater detail in Chapter 5.

### 3.5.3 Profiling Information

In order to make realistic cost predictions and select an efficient implementation, some estimates of the input data sizes and the costs of sequential functions are required. This information could be obtained by incorporating profiling and type checking capabilities in the analyser. The current implementation of the analyser does not include these capabilities, and the user specifies this information.

In order to estimate communication costs, information such as start-up time, denoted by $K_0$, the bandwidth of the communication channel, denoted by $K_1$, and the size of the data to be communicated is required. $K_0$ and $K_1$ are machine-specific parameters. A linear model of communication is assumed. The size of the data to be communicated, and therefore the communication cost, also depends on the number of list elements and the size of each element. The size of the input list could be obtained by using profiling information. The size of each base element depends on its type and this could be deduced by a type-checker. This is made possible because, as already discussed, polymorphic functions are not permitted. These issues are discussed more elaborately in Chapter 5.

### 3.5.4 The Code Generator

The *code generator*, as shown in Figure 3.1 would generate code for the target parallel machine with appropriate communication constructs inserted. However, a fully-fledged code generator has not been implemented. To provide preliminary evidence of the performance of the HOPP model and to assist in program development, some support is available in the form of a library of functions. This library contains the code for the various recognised functions, and also code for performing various types of communications on a particular parallel machine topology. The code in this library is used by all the problems in the performance study. The actual calls to the functions are at present generated by hand. The use of the same code ensures that performance figures for the different examples can be sensibly compared.

## 3.6 Summary

Many of the recognised functions are also used in the skeletons approach [Col89, Col88, Col87, D$^+$93, HH93]. As in the case of the skeletons, HOPP aims to provide a platform for developing parallel programs where the programmer is not explicitly responsible for parallelism. However, the idea is to be able to express programs that may not entirely match existing skeletons. In that sense, HOPP can be viewed as a more fine-grained approach to parallelising programs. Also, the emphasis is on cost analysis of programs, in an effort to obtain cost-effective parallel implementations.