HOPP - A HIGHER-ORDER PARALLEL PROGRAMMING MODEL

ROOPA RANGASWAMI
Department of Computer Science
The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ
Scotland, U.K.
ror@dcs.ed.ac.uk

ABSTRACT. The efficient programming of parallel computers is still a difficult task. This paper focuses on studying methods for expressing parallelism in programs without making the programmer explicitly responsible for parallelism. Parallel programming using a set of useful implicitly-parallel constructs is considered. These constructs are borrowed from FP and the Bird-Meertens approach. Programs are analysed statically, using an analytical cost model which selects a cost-effective implementation for a chosen architecture. A brief overview of the model is presented and its operation is demonstrated with the aid of two case studies.

KEYWORDS. Parallelisation Techniques, Functional Programming, Bird-Meertens Formalism, Higher-Order Functions, Compile-time Analysis, D-M MIMD Machines.

1 INTRODUCTION

Programming parallel computers is still a complex task. The programmer is often responsible for identifying the parallelism, sharing the work-load amongst the processors, and handling interprocess communications. The resulting programs are not particularly portable, being specific to a particular architecture.

HOPP is an approach to parallel programming based on an implicitly-parallel functional language. Functional languages have advantages such as referential transparency and abstraction from detail. Referential transparency ensures that a given expression evaluates to the same value irrespective of the order of evaluation. Abstraction from detail is obtained by the use of *higher-order* functions. A function can be an argument to another function as well as the result of a function application. A function that has one or both of these properties is called a *higher-order* function. This ability to abstract away from detail hides the complexities of parallel programming from the programmer. However, one of the problems with functional languages is their poor efficiency of implementation compared to imperative languages. The main reason for this is the use of lists as the main data structure. The sequential implementations suffer from overheads arising from creation and destruction of lists and garbage collection. Also, list access is linear in its length as opposed to constant time access in the case of the array. In addition, parallel implementations using list data structures incur significant overheads due to *packing* and *unpacking* when communicated between processors.

HOPP is based on constructs borrowed from the Bird-Meertens Formalism [2], and FP [1]. It incorporates a basic set of higher-order functions that perform general-purpose operations on lists. These functions are already part of, or can be defined in, most functional languages. They are inherently parallel and are referred to as *recognised functions*. Since the behaviour of each of these 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. The main limitation of this approach is that the analysis can only handle *regular* problems. A *regular* problem is one whose behaviour does not depend on the actual input values. The set of higher-order functions themselves impose a degree of regularity on the problem. It would be difficult to express problems requiring irregular communications, using the set of available recognised functions. This means that many irregular problems cannot be parallelised using this approach. In the extreme case where a program does not contain any occurrences of recognised functions, no parallel implementation can be realised. This forces the programmer to remain within the fixed repertoire of available functions in order to obtain parallelism.

HOPP aims to generalise the Skeletons approach considered in [3],[4], [5], [6]. Algorithmic Skeletons capture common computational forms. A skeleton is an abstraction of a well-known parallel computational form, e.g., *divide-and-conquer*. Each application program must match one or more skeletons in order to be exploited for parallelism. The implementation scheme provides an efficient implementation for the program. The method considered in this paper aims to provide a more general basis for writing programs. The idea is to be able to express programs that may not necessarily match one or more skeletons. An approach similar to the one in this paper, is also considered in [8],[9]. However, the cost measures employed there are simplistic particularly when accounting for communication costs. We describe a *hierarchical* cost model, i.e., parallel implementations are considered for recognised functions and their arguments. The method also aims to minimise the communication costs of implementations.

2 OVERVIEW OF THE SCHEME

A program in HOPP is expressed as a *sequence* of phases. Each phase comprises of one or more occurrences of one or more 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. User-defined functions only have a sequential implementation. A knowledge of the target machine topology is essential in order to select the implementation associated with that topology

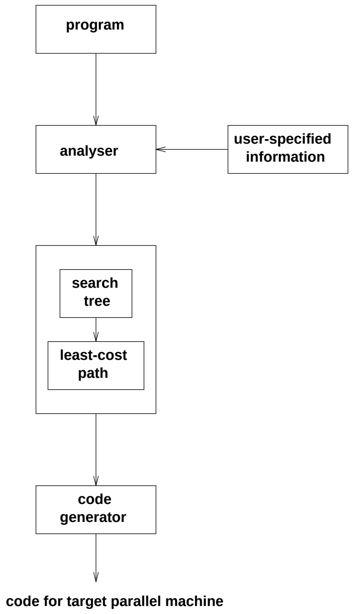

Figure 1: The Analysis and Implementation Scheme

for the recognised function. Each implementation attempts to make optimum use of the machine connectivity in an effort to reduce communications overhead. The hypercube, 2-D torus and binary tree topologies have been studied. However, this paper only discusses the scheme in the context of the hypercube topology.

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.

The basic set of functions in the Bird-Meertens Formalism has been extended to incorporate a number of *new* functions. It has been shown that all these functions can be expressed in terms of one or more of the existing functions. They have been included as recognised functions in their own right because they are found to be useful in many common problems. Also, the scheme provides a more efficient implementation for them as compared to the implementation that would be obtained if they were expressed in terms of the basic set of functions.

The basic outline of the scheme is depicted in Figure 1. The analyser constructs a *parse tree* representing the program. Each branch in the tree corresponds to a phase of the program. A cost analysis is carried out on the program.The cost of a program comprising of n phases is given by:

$$Cost = \sum_{i=1}^{n} Cost_i$$

where, $Cost_i$ is the cost of phase i. The cost of a phase depends on the nature and number of recognised functions in that phase and also the parallel implementation selected for that phase, on a given number of processors. In our 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 target machine (in this case, a d-dimensional p-processor hypercube). 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 outermost function on p processors.
- A parallel implementation for the innermost function p processors.
- A parallel implementation for both functions. The d-dimensional 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 processors and the inner function is evaluated in parallel across 2^{d-k} processors.

A similar argument can be applied to phases containing three or more recognised functions. However, parallel implementations are only considered up to three recognised functions in the phase. Any recognised functions below this are just implemented sequentially. Hence, a phase can have at most eight possible implementations (including a sequential one).

The costs associated with all the implementations for each of the phases are estimated and a search tree is constructed. The most efficient implementation for the program corresponds to the least-cost path in the search tree. The code corresponding to this least-cost path can then be generated and executed on the parallel machine. The search tree grows exponentially with the number of phases in the program. In this paper, only problems that comprise of a few phases are discussed, so this is not a serious issue. However, for bigger problems some heuristics for pruning the search tree would have to be considered.

In order to predict costs fairly accurately and select efficient implementations, some estimate of the input data sizes and the costs of sequential functions are required. This could either be obtained by profiling or from the user. Since the system does not incorporate a profiler, the user is required to specify such information. In order to estimate communications costs, information such as start-up time K_0, the bandwidth of the communication channel 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 depends on the number of list elements and the size of each element. If the list size is unknown at compile-time, the user is required to estimate it as a function of the number of processors p. Simple comparison metrics are used, such as, list size $\gg p$, list size $\approx p$, etc. The size of each base element depends on its type and this could be deduced by the compiler. However, programs in functional languages can be written at a very high level of abstraction. A program could apply to a whole group of list types and the exact type information will only be known at runtime. In such cases, the user will be required to specify type information at compile-time.

It is important to realise that the analysis scheme does not account for costs arising from low level operations such as memory access. This would involve a very low level of detail for the analyser, and may make it very machine-specific. The selection of the least-cost implementation depends only on cost comparison and hence absolute costs are not of consequence. List processing costs are however, accounted for by the model since this forms a substantial overhead in functional languages. This includes costs that 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.

The *code generator* shown in Figure 1 would generate C-code for the target parallel machine with appropriate communication constructs inserted. However, this 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 the hypercube topology. The hypercube communication functions are based on [7]. The code in this library is used by all the example problems. The actual calls to the functions are at present generated by hand. The use of the same code ensures that performance figures for different examples can be sensibly compared.

3 CASE STUDIES

This section illustrates the method used by HOPP, by considering two examples - matrix multiplication and merge sort. For each example, the problem is first expressed in the style just advocated - i.e., as a sequence of phases, with each phase comprising one or more recognised functions. The implementations considered and the costs computed by the analyser are examined. The results of implementing the most efficient implementation selected by the analyser on the Meiko Computing Surface (a transputer-based machine), are presented. The next best implementation selected by the analyser is also implemented, in order to verify the correctness of the model. Finally, a comparison of the predicted and experimental results is presented. Since the Meiko Computing Surface is transputer-based, a processor has at most four links. Hence, the maximum dimension of the hypercube considered for experimental purposes is four.

3.1 Definitions and costs for some functions

The extended set of recognised functions contains about twenty-five different functions. However, only a few of these are used by the examples discussed in this paper. The definitions for these functions are given below. Informal ML-style [10] notation is used. The cost of implementing each function on a d-dimensional, p-processor hypercube is also given. In the following discussion, C_f represents the cost of the function f. As discussed previously, this cost is to be specified by the user or can be obtained by profiling techniques. Also, n represents the length of the input list. It is assumed that the input list is distributed across the p processors, $\lceil \frac{n}{p} \rceil$ per processor.

- **map** - applies some function to every element of a list.

  map f [ ] = [ ]
  map f (x::xs) = (f x) :: map f xs

The cost of **map** is given by : $\lceil \frac{n}{p} \rceil \ C_f$.
The function **map2** is similar to **map**, but operates on two input lists of equal length. Its cost is the same as the cost of **map**.

map2 f [ ] [ ] = [ ]
map2 f (x::xs) (y::ys) = (f x y) :: map2 f xs ys

- **fold** - combines the elements of a list, taken two at a time, using some binary operator *f*. When a call is made to the **fold** function, a must be the identity of the binary operator. At each step of the computation, a contains the partial result of the **fold** operation.

  fold f a [ ]= a
  fold f a (x::xs) = fold f (f a x) xs;

  The scheme assumes that the argument function f is associative. Only if this condition is met, can **fold** be implemented in parallel.

  Each processor performs the **fold** on its local elements. Then, the partial results are combined globally to obtain the final result. The first step involves $\lceil \frac{n}{p} \rceil - 1$ applications of the function f. The second step involves d communications and d applications of the function f. Hence the cost of **fold** is given by : $C_f(\lceil \frac{n}{p} \rceil - 1) + d(T_{com} + C_f)$. T_{com} represents the cost of communicating the partial result from one processor to its neighbour. This cost depends on the size of the partial result being communicated.

- **cross-product**. Based on the order of evaluation, we define two functions for performing the **cross-product** operation. The basic operation is the same in both cases, but the order in which they are performed is different, causing the ordering of elements to be changed in the result.

  - **r_cross_product**

    $$\begin{aligned}\text{r\_cross\_product f } [a_1,a_2,\ldots,a_m]\ [b_1,b_2,\ldots,b_n] &= [[(\text{f } a_1\ b_1), (\text{f } a_1\ b_2), \ldots, (\text{f } a_1\ b_n)],\\ &\quad [(\text{f } a_2\ b_1), (\text{f } a_2\ b_2), \ldots, (\text{f } a_2\ b_n)],\\ &\quad \vdots\\ &\quad [(\text{f } a_m\ b_1), (\text{f } a_m\ b_2), \ldots, (\text{f } a_m\ b_n)]]\end{aligned}$$

  - **c_cross_product**

    $$\begin{aligned}\text{c\_cross\_product f } [a_1,a_2,\ldots,a_m]\ [b_1,b_2,\ldots,b_n] &= [[(\text{f } a_1\ b_1), (\text{f } a_2\ b_1), \ldots, (\text{f } a_m\ b_1)],\\ &\quad [(\text{f } a_1\ b_2), (\text{f } a_2\ b_2), \ldots, (\text{f } a_m\ b_2)],\\ &\quad \vdots\\ &\quad [(\text{f } a_1\ b_n), (\text{f } a_2\ b_n), \ldots, (\text{f } a_m\ b_n)]]\end{aligned}$$

    The cost of **r_cross_product** is given by : $\lceil \frac{m}{p} \rceil nC_f$. The cost of **c_cross_product** is given by : $\lceil \frac{n}{p} \rceil mC_f$. The second list would have to be broadcast to all the processors.

- **split** - splits a given list into a specified number of sublists.

  split 1 $[x_1, x_2, \ldots, x_n] = [[x_1, x_2, \ldots, x_n]]$
  split k $[x_1, x_2, \ldots, x_n] = [[x_1, x_2, \ldots, x_{\lceil \frac{n}{k} \rceil}], \ldots, [x_{(k-1)\lceil \frac{n}{k} \rceil + 1}, \ldots, x_n]]$

  split only involves list re-arrangement costs. A list of size n is re-arranged to a list of lists, with $(k-1)$ sublists of size $\lceil \frac{n}{k} \rceil$ and the last sublist of length $n - (k-1)\lceil \frac{n}{k} \rceil$. These list processing costs are calculated by the analyser, based on the nature of the input list. For the purposes of this paper, the cost of splitting an n-element list into k parts, is represented by $S_{n,k}$.

- **composition**. The **composition** operation is represented by $\circ$ and corresponds to the *sequence* of phases in a program. The terms composition and sequence will henceforth be used interchangeably.
  (f $\circ$ g) x = f (g x)
  The cost of (f $\circ$ g) is given by : $C_f + C_g$. Occurrences of the form f (g x) will be replaced with (f $\circ$ g) x before the cost analysis is carried out.

3.2 Matrix multiplication

This section discusses the well-known problem of multiplying two matrices $A_{m \times n}$ and $B_{n \times k}$, resulting in the matrix $C_{m \times k}$.

3.2.1 Program code The problem can be expressed as a composition of two phases as shown below. The recognised functions are depicted in bold face and informal ML-style notation is used.

fun mat_mult times plus A B = **map** (**map** (**fold** plus 0)) $\circ$
r_cross_product (**map2** times) A B^T;

where,
fun plus a b = a + b;
fun times a b = a * b;
B^T represents the transpose of B

The transpose is performed sequentially, in order to improve program performance.

3.2.2 The analysis The analyser constructs the parse tree represented in Figure 2 for the problem. The root node represents **composition**. For phase one,the analyser computes the costs for a sequential implementation and three parallel implementations, corresponding to the two recognised functions in the phase. On a p-processor hypercube, if **r_cross_product** is implemented on p_1^1 processors and **map2** is implemented on p_2^1 processors, $p_1^1 \leq p$, $p_2^1 \leq p$ and $p_1^1 p_2^1 = p$, the cost of phase one is as follows : (see section 3.1)

$$C_{p1} = C_{com} + \lceil \frac{m}{p_1^1} \rceil \lceil \frac{n}{p_2^1} \rceil k C_{times}$$

C_{com} represents the costs incurred in distributing the initial input lists across the p processors. C_{times} represents the cost of the sequential function *times*.It may be noted that the

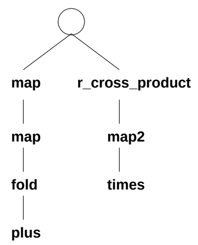

Figure 2: The Parse Tree for Matrix Multiplication

result of phase one is a list of list of lists of size $(m \times n \times k)$. Phase two contains three recognised functions. The analysis results in the costs for a sequential implementation and seven different parallel implementations. If the outer **map** is implemented on p_1^2 processors, the inner **map** is implemented on p_2^2 processors and the **fold** is implemented on p_3^2 processors, $p_1^2 \leq p$, $p_2^2 \leq p$, $p_3^2 \leq p$ and $p_1^2 p_2^2 p_3^2 = p$, the cost of phase two is computed as follows :

$$C_{p2} = R_{com} + \lceil \frac{m}{p_1^2} \rceil \lceil \frac{n}{p_2^2} \rceil C_{plus} (\lceil \frac{k}{p_3^2} \rceil - 1) + d(T_{com} + C_{plus})$$

R_{com} represents the cost incurred in re-arranging the results of phase one to suit the implementation of phase two, $d = \log_2 p_3^2$, T_{com} is the cost incurred in communicating the partial result of the **fold** on one processor to a neighbouring processor.

Two sets of matrices $A^1_{32\times32}$, $B^1_{32\times32}$ and $A^2_{64\times32}$, $B^2_{32\times16}$, were considered for analysis. In both the cases, the following implementations were selected as the best (i.e., least-cost) and second best parallel implementations respectively, for $p = 2, 4, 8, 16$.

1. Best - For the two phases, $p_1^1 = p$, $p_2^1 = 1$ and $p_1^2 = p$, $p_2^2 = 1$.This corresponds to a parallel implementation for **r_cross_product** in the first phase and a parallel implementation for the outer **map** in the second phase.

2. Second Best - For $p = 2$, it is clear that only one function in a phase can be implemented in parallel. For the second best implementation, **map2** is implemented in parallel in the first phase and **fold** is implemented in parallel in the second phase. For $p = 4, 8, 16$, the following implementation was selected as the second best. In phase one, $p_1^1 = 2$, $p_2^1 = \frac{p}{p_1^1}$. In phase two, $p_1^2 = 2$, $p_2^2 = 1$ and $p_3^2 = \frac{p}{p_1^2}$. This corresponds to a parallel implementation for **r_cross_product** on p_1^1 processors and **map2** on p_2^1 processors in the first phase. In the second phase, the outer **map** and the **fold** are implemented in parallel on p_1^2 and p_3^2 processors respectively. The inner **map** is implemented sequentially.

3.2.3 The results The theoretically predicted values are plotted along with those obtained experimentally in Figure 3, for both implementations 1 and 2. In each of the graphs, the *trend* predicted by the model is also obtained experimentally. The experimental curves are above the theoretical ones. This is because the model does not account for low level costs that are incurred by a practical implementation. For both sets of matrices, the optimal number of processors predicted is verified experimentally. Also, the theoretical and

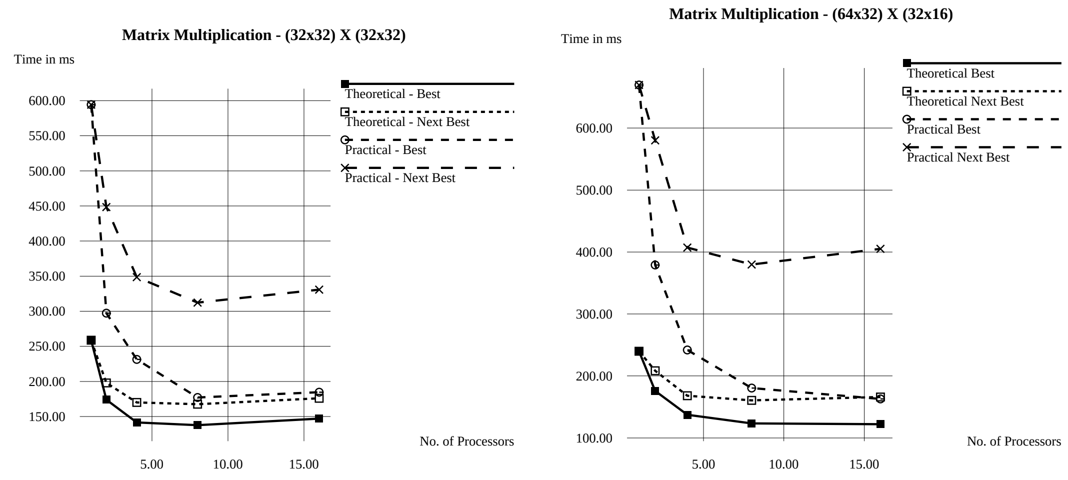

Figure 3: The Results for Matrix Multiplication

experimental curves in implementation 1 are much closer than the respective curves in implementation 2. The same sequential code is executed by both implementations. However, implementation 2 involves more communications (such as scattering and broadcasting) for initial data distribution. Also, the parallel implementation of **fold** in phase two involves communications. An implementation involving more communications appears to perform worse than predicted. This is probably due to the method used to estimate K_0 and K_1, the machine-specific communication parameters. Communication costs were measured for different data sizes. For each data size, an average cost over a large number of such communications was considered. K_0 and K_1 were then obtained by a curve fitting technique. In the case of a real program, the cost of just one instance of a communication is measured. The processors also probably do not operate in lock-step. Communication routines such as *scatter* and *broadcast* assume a number of overlapping nearest-neighbour communications. These may not necessarily overlap in practice. This probably explains the larger distance between the theoretical and experimental curves for implementation 2.

3.3 Merge sort

This section discusses merge sort. A list of integers is to be sorted in ascending order. The strategy adopted is to sub-divide the list into a list of lists, with each sublist containing two elements. The two elements in each sublist are sorted and the sorted sublists are then merged, two at a time, to obtain a fully sorted list. For the sake of simplicity, the number of elements in the input list is assumed to be a power of 2.

3.3.1 Program code The problem can be expressed as a sequence of three phases as shown below. The recognised functions are again depicted in bold face. The function **g_fold** will be explained shortly.

fun sort n xs = **g_fold** (merge [ ]) ∘ **map** msort ∘
split (n/2) xs;

where,

```
n is the length of the list
fun msort [x,y] = if(x ≥ y) then [y,x] else [x,y];
fun merge [ ] [ ] = [ ]
    merge xs [ ] = xs
    merge [ ] ys = ys
    merge (x::xs) (y::ys) = if (x ≤ y) then
                                x :: merge xs (y::ys)
                            else y :: merge (x::xs) ys;
```

3.3.2 The analysis In this case, all three phases have only one recognised function each. Hence, only two implementations are possible for each phase - one parallel and one sequential. The cost of the function *msort* is a constant, but the cost of *merge* is proportional to the sizes of the two input lists that are being merged. In this case, the scheme allows the user to specify a cost function for the sequential function *merge*. This cost function is used by the analyser to estimate implementation costs.

The analyser constructs the parse tree representing the program. For phase one, the cost of **split** is given by $S_{\frac{n}{p},\frac{n}{2p}}$. The cost for the sequential implementation is obtained by making $p = 1$. At the end of phase one, each processor contains a list of lists of size $(\frac{n}{2p} \times 2)$. Phase two only involves a **map** and its cost is given by $\frac{n}{2p}C_{ms}$, where C_{ms} represents the cost of the function *msort*. In phase three, after each *merge* operation, the size of the resulting list is the sum of the sizes of the two lists being merged. As the **fold** progresses, the size of the intermediate result increases. After each processor executes the **fold** locally, the number of elements on each processor is $\frac{n}{p}$. In the first step of the combination of partial results, two lists each of size $\frac{n}{p}$ are merged to produce a list of size $\frac{2n}{p}$. The second step involves the merging of two lists each of size $\frac{2n}{p}$ and so on. Let the cost of merging two lists of sizes m and n be represented by $f(m,n)$. Using the formula for the cost of **fold** in section 3.1, the cost of phase three is as follows :

$$\sum_{i=1}^{\frac{n}{2p}-1} f(2i,2) + \sum_{i=0}^{d-1}(T_{com}^{2^i\frac{n}{p}} + f(2^i\frac{n}{p}, 2^i\frac{n}{p}))$$

In order that the analyser accounts for the increasing size of the emerging result while calculating communications costs, the user is required to specify the **fold** operation by the use of **g_fold**. In the case of matrix multiplication the size of the emerging result is a constant. The computation in that case is therefore specified by an ordinary **fold**. Syntactically there is no difference between **fold** and **g_fold**. **g_fold** provides extra information to the analyser, enabling it to predict costs more accurately.

Two lists of sizes 512 and 1024 respectively, were considered for analysis. In both cases the following implementations were selected as the best (i.e., least-cost) and second best implementations respectively, for $p = 2, 4, 8, 16$.

1. Best - Implement each recognised function in parallel in all three phases on p processors.
2. Second Best - In phase one implement the **split** sequentially on a single processor.

Then scatter the result to implement the two recognised functions in the next two phases in parallel on p processors.

3.3.3 The results The results for merge sort are plotted in Figure 4. Once again the trend predicted by the model is obtained experimentally. For the input list of size 512,

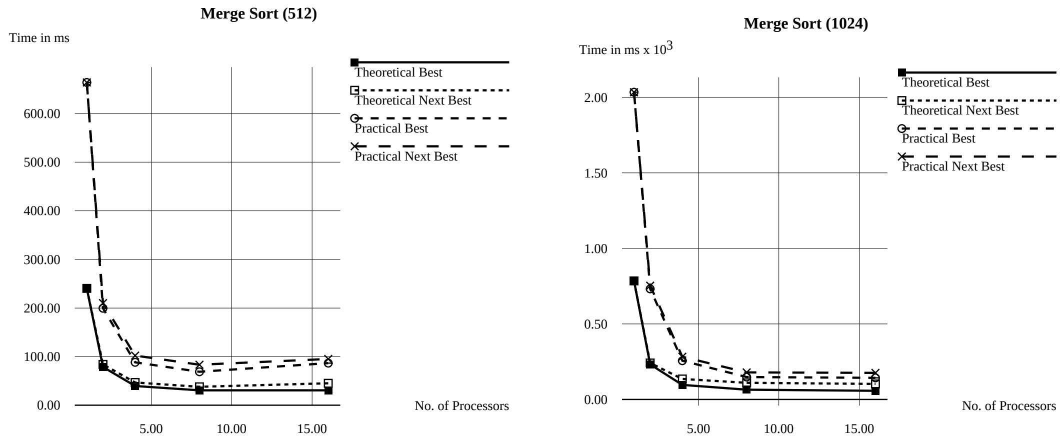

Figure 4: The Results for Merge Sort

the optimal number of processors is predicted to be 8 for both implementations. This is also obtained experimentally. For the list of size 1024, a more effective implementation was obtained with 16 processors both theoretically and experimentally.

In the case of merge sort, fewer communications are involved in the initial distribution of data, as compared to either implementation of matrix multiplication. A closer match between the predicted and experimental results is obtained.

4 CONCLUSIONS

The paper discussed two application programs that were expressed using the HOPP model. The constructs used to express the applications had predefined parallel implementations and costs. An analyser computed the most cost-effective implementation for each of the programs. These were implemented and the predicted and experimental implementations were found to match fairly well. The two programs were of a *regular* nature and it was therefore possible to express them in terms of the existing constructs. Irregular problems such as quicksort that depend on the actual input data values cannot be effectively parallelised using this scheme. This is a limitation of the HOPP model and arises due to the use of compile-time techniques to predict program performance.The advantages of the model include architecture-independent programs and simplicity of programming parallel machines.

Future work needs to concentrate on testing the expressiveness of the model by considering more examples. The code generator needs to be implemented. A profiler could be incorporated in the system, in order to reduce the amount of information required from the user. It would be interesting to study the HOPP model on shared-memory machines.

Acknowledgements

I would like to thank Murray Cole for supervising this work. Thanks are also due to The Edinburgh Parallel Computing Centre for providing me with the parallel machine facilities. This work was partly funded by the Overseas Research Studentship awarded by the British Council.

References

[1] J.Backus. Can Programming be liberated from the Von Neumann stye : a Functional style and its algebra of programs. In *Comm. ACM*, Vol. 21, No.8 Aug. 1978, 613-41.

[2] R.S.Bird. Algebraic Identities for Program Calculation. In *The Computer Journal, Vol.32, No.2* 1989, 122-126.

[3] M.Cole. Algorithmic Skeletons : Structured Management of Parallel Computations. *Research Monographs in Parallel and Distributed Computing* Pitman 1989.

[4] M.Cole. Higher-Order Functions for Parallel Evaluation. In *Proceedings of the 1988 Glasgow Workshop on Functional Programming* August 1988, 8-20.

[5] J.Darlington et al. Parallel Programming Using Skeleton Functions. In *PARLE '93 Parallel Architectures and Languages Europe*. 5th International PARLE Conference, Munich, Germany, June 1993, 146-160.

[6] H.Stolze, H.Kuchen. Parallel Functional Programming Using Algorithmic Skeletons. In *Parallel Computing : Trends and Applications*. Proceedings of the International Conference ParCo93, Grenoble, France, September 1993, 651-654.

[7] Y.Saad, M.H.Schultz. Data Communications in Hypercubes. In *Journal of Parallel and Distributed Computing* 6, 1989, 115-135.

[8] D.B. Skillicorn. Parallelism and the Bird-Meertens Formalism. *Internal Report, Department of Computing and Information Sciences* Queens University, Kingston, Canada, April 1992.

[9] D.B.Skillicorn and W.Cai. A Cost Calculus for Parallel Functional Programming. *Internal Report, Department of Computing and Information Sciences* Queens University, Kingston, Canada, May 1992.

[10] M.Tofte. Four Lectures on Standard ML. *Technical Report, ECS-LFCS-89-73*, University of Edinburgh, 1989.