A SCHEME FOR NESTING ALGORITHMIC SKELETONS

Mohammad Hamdan*, Greg Michaelson and Peter King

Department of Computing and Electrical Engineering,
Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS
{hamdan,greg,pjbk}@cee.hw.ac.uk

Abstract. A scheme for arbitrary nesting of algorithmic skeletons is explained which is based on the idea of groups in MPI. Two skeletons were developed which run in a nested mode: a binary divide and conquer and a process farm for a parallel implementation of **fold** and **map** HOFs respectively. An Example showing various cases for nesting the two skeletons is presented. The experiment was conducted on the Fujitsu AP1000 parallel machine.

1 Introduction

It is well known that parallelism adds an additional level of difficulty to software development. Following Cole's characterisation[1], algorithmic skeletons have been recognised widely as a valuable basis for parallel software construction. A skeleton abstracts a control structure which may be instantiated subsequently with specific functions to carry out specific tasks. Therefore, the encapsulation of parallel algorithms into skeletons is a promising approach to high-level specification of parallel algorithms. Normally, functional programming languages are used as a framework for representing skeletons as higher-order functions.

The goal of this work is to allow arbirtray nesting of skeletons and automatically generate parallel code for the program prototype. This would help the programmer in expressing more complex parallel algorithms and thus having larger parallel applications. Although it is arguable that arbitrary nesting of algorithmic skeletons is not needed we still think there is a need to find a general solution to the problem of nesting.

The paper is organised as follows. Section 2 surveys the work done on algorithmic skeletons. The programming model used is outlined is Section 3. Section 4 presents the scheme used to achieve arbitrary nesting. The test example and results are shown in Section 5 and finally Section 6 concludes.

2 Skeleton-Oriented Programming

Algorithmic Skeletons is one of the general models [2] for programming parallel machines which was introduced by Murray Cole [1] in 1989. This model provides an alternative approach which aims for high abstraction and portability across different architectures.

* Author for correspondence.

The idea is to capture common patterns of parallel computation in Higher Order Functions (HOFs). As we know, HOFs are a natural part of functional languages and a programmer can easily manage and reason about them. This results from the fact that parallelism is restricted only to a small set of HOFs. Also, parallelising compilers are capable of exploiting efficiently their implicit parallelism [3].

The **algorithmic skeletons** major advantage is portability of parallel programs written using this approach. This results from the separation of meaning and behaviour for each skeleton which was identified by **Darlington et al** [4]. A HOF in a functional language can be used to express the declarative meaning for a skeleton which could have more than one implementation. Therefore, the declarative meaning is independent of any implementation and hence any behavioural constraints [5].

Campbell [6] has proposed a general classification of algorithmic skeletons which is used to outline some of the well known skeletons in the following sections.

2.1 Recursively Partitioned

A recursively partitioned skeleton works by generating subordinate problems as a result of dividing the original problem. These generated problems will also be divided further in an attempt to solve them. Two well known algorithms that belong to this category are quick-sort and least-cost search. **Rabhi** [7] gave a HOF describing this skeleton:

```
rp$ ind solve divide combine prob
         | ind prob  = solve prob
         | otherwise = combine prob
                        (map (rp$ ind solve divide  combine)
                             (divide prob))
```

where:

- ind prob: returns true if the problem **prob** is indivisible.
- solve prob: solves an indivisible instance of a problem **prob**.
- divide prob: divides the problem **prob** into sub-problems.
- combine prob sols: combines the solutions **sols**.

Darlington et al [8] have presented a HOF for the divide and conquer (DC) skeleton. Similar to the recursively partitioned skeleton, the idea is to split larger tasks into sub-tasks and combine the results obtained by solving the sub-tasks independently. Their HOF for DC is as follows:

$$DC :: (\alpha \to Bool) \to (\alpha \to \beta) \to (\alpha \to [\alpha]) \to ([\beta] \to \beta) \to \alpha \to \beta$$

```
DC t s d c x | t x       = s x
             | not (t x) = (c . map (DC t s d  c) . d) x
```

where:

- t x: tests if a problem **x** is trivial.
- s x: solves a problem **x**.
- d x: divides a larger task **x** into sub-tasks.
- c ((map DC ...) x): combines the sub-results to produce the main result.

Cole [1] has also presented a HOF for his fixed degree divide and conquer (FDDC) skeleton. This is a special form of the general divide and conquer skeleton where the number of sub-problems to be divided are fixed in advance.

$$D_C : (prob \to bool) \to (prob \to [prob]) \to$$

$$([sol] \to sol) \to (prob \to sol) \to (prob \to sol)$$

```
D_C indivisible split join f = F
         where F P   = f P, if indivisible P
                     = join_k (map F (split_k P)), otherwise
```

Feldcamp et al [9] have developed a divide and conquer skeleton which was integrated in the program development environment **Parsec** [10].

2.2 Task Queue

Cole [1] has not presented a HOF for the **task queue** skeleton. However, a HOF for the **process farm** skeleton (discussed below) can be regarded as a special form of **task queue**. Algorithms that belong to **task queue** have workers in a work pool where each worker will check if tasks are available and then group the available one. Next, the workers will execute the tasks which might result in creating other tasks. The generated tasks will be added to the queue. The algorithm terminates when no more tasks are available and all processors are inactive. When tasks are executed they access a shared data structure that represents the problem to be solved. The **task queue** can have a number of queuing disciplines; stack, FIFO queue, unordered heap and strictly ordered queue.

Darlington et al. [8] have presented a HOF for the **process farm** skeleton which is a special form of the **task queue** skeleton. The **process farm** skeleton represents simple data-parallelism. A function is applied to a list of data. Parallelism arises by utilising multiple processors to apply the function to different parts of the list. Their HOF is as follows:

$$FARM :: (\alpha \to \beta \to \gamma) \to \alpha \to ([\beta] \to [\gamma])$$

```
FARM f env = map (f env)
```

Bratvold [11], **Busvine** [12] and **Sreekantaswamy** [13] have developed a process farm skeleton as part of their work.

2.3 Systolic

The **systolic** skeleton (which is the general form of pipeline-like skeletons) consists of a number of stages where parallelism is exploited by operating on a flow of data that passes through the various stages of the pipeline.

Darlington et al [4] have presented a HOF for the pipe skeletons, which is:

$$PIPE :: [[\alpha] \rightarrow [\alpha]] \rightarrow [\alpha] \rightarrow [\alpha]$$

```
PIPE = foldr1 ( o )
```

Bratvold [11] has done work on the pipeline skeleton.

2.4 Skeleton Systems

Other researchers have extended work on simple skeletons to include in a complete system for parallel programming. The idea is to integrate all skeletons into a single system and use them to express parallel programms. The main work in this area is the work of **S. Pelagatti** [14] where she has participated in developing an explicit parallel programming language called P^3L [15,16].

Also, **R. Rangaswami** [17] has developed a parallel programming model called **HOPP** for skeleton-oriented programming. **HOPP** stands for Higher-Order Parallel Programming. The model consists of three parts: the **program** model, the **machine** model and the **cost** model. The program model is a composition of nested instantiations of recognised and user-defined functions [1]. Each stage in the pipeline resulting from the composition is referred to a **phase** of the program. The machine model targets architectures for the programs which include hypercube, 2-D torus, linear array and tree. The cost model determines the cost of executing a recognised function on a given architecture. In her system nesting of skeletons was limited only to the first 3 levels and it was not automated as the code had to be generated manually.

The work of **H. W. To** [18] was about optimising the combinations of algorithmic skeletons. A language for combining skeletons was proposed and a set of **primitive skeletons** was chosen. The primitive skeletons were based on the operators of parallel abstract data types (PADTs)[2]. This choice was based on the observation that many highly parallel applications exploit parallelism from a large data structure. Two PADTs were described: restricted lists (**rlist**s) and **arrays**. A number of combining skeletons were proposed where each skeleton captures a common pattern of control flow. For example, the compose ($\circ$) function [3] which takes the output of one skeleton and pass it as the input of anther skeleton. Another two combining skeletons are **iterateFor** and **iterateUntil** which actually extend composition where the same skeleton might iterate over a data structure. Also it is possible to combine binary skeletons using the compose-2 ($\circ\circ$) function. In cases where it is necessary to apply more than one skeleton to an instance of a data structure the split ($\triangle$) skeleton could be used.

[1] All of the recognised functions work on lists.
[2] A PADT is an aggregate data structure and a set of parallel operators with which to manipulate it.
[3] Note that we are talking about composing skeletons here and not functions.

3 The Programming Model

Our concern it to investigate techniques for arbirtary nesting of algorithmic skeletons. Therefore, we have chosen a parallel programming model which has the following features:

- Provides the programmer with a fixed set of primitives that perform general-purpose operations on lists. These primitives are inherently parallel and the programmer uses them to express programms in terms of compositions of these sete of functions along with user-defined functions.
- Abstracts away from the target architecture in order to insure portability.
- Hides low-level issues like load balancing, process mapping and inter-process communication from the programmer.

Next we will outline the language used for research into nesting skeletons.

3.1 EKTRAN

EKTRAN[4] is a sub-set of **FP** which was introduced by John Backus in the late 1970s [19]. Its main features are: (1) It is a simple functional progrmming language with syntax similar to SML [20], (2) It provides a set of basic functions which can be combined to build other functions and (3) It is an implicit language with no control over process mapping, data distribution and inter-process communication. The language is so simple as it is needed only as a programming model which allows combining and nesting of basic primitives.

4 Overview of the Scheme

The proposed scheme for nesting algorithmic skeletons uses the principle of groups in the message passing interface (MPI) [21]. In general, if a system wants to nest skeletons then it should manage the following:

- Assign a number of processes to a skeleton.
- Allow the outer most skeleton to be able to assign processes to inner skeleton. This is needed as the idea of nesting is to run two or more skeletons in parallel at the same time.
- Manage communication between the different levels of nesting and within each separate skeleton.

Fortunately, MPI provides routines for managing groups where processes belong to a group. If a group contains **n** processes then its processes are identified within the group by ranks which are used by MPI to label processes in groups and they are integers from **0** to **n - 1**.

MPI has two routines to manage groups: split and create a group. The former takes a group and splits (divides) it into sub-groups. The division is based on a key value which we call **split**. The later creates a new group from an existing group. This has an advantage

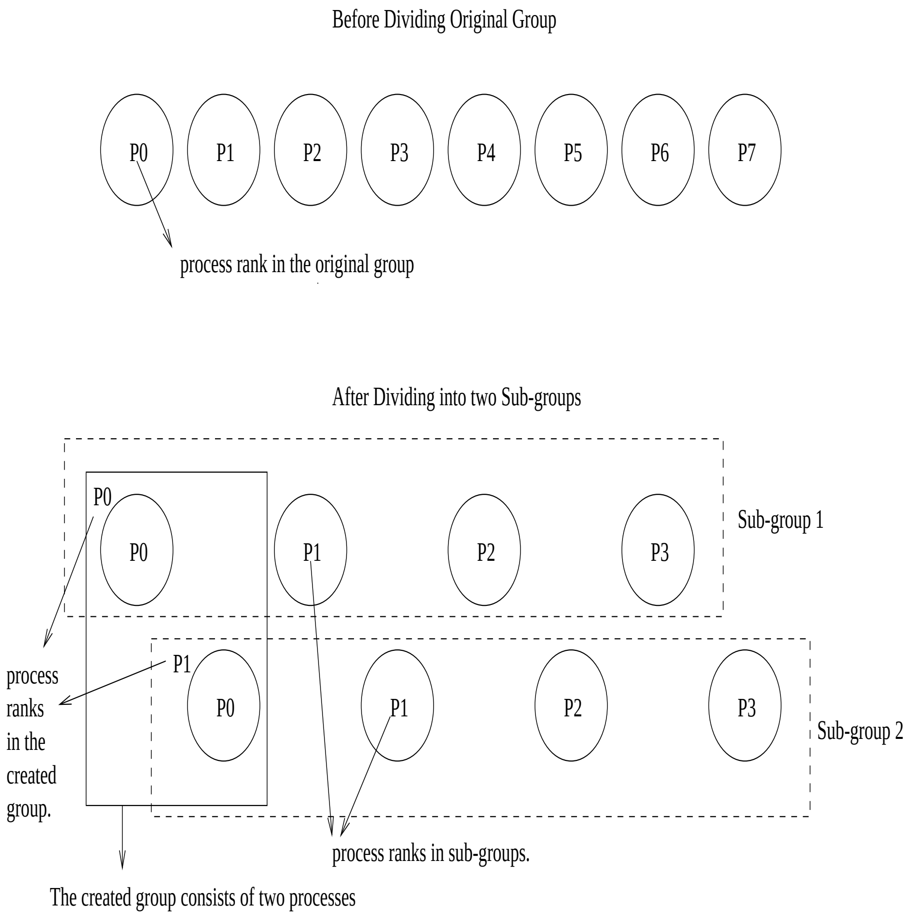

Fig. 1. Dividing and creating groups from an existing group.

of deciding which processes to include in the group. The process of dividing and creating groups at run time makes our system to be dynamic and not static.

Figure 1 shows how a group could be divided into sub-groups and how a new group is created from existing groups.

Each skeleton has its own group of processes. When it is executed it has control over a fixed number of processes. If its function argument is another skeleton then it is running in a nested mode and one of the skeleton arguments which is called **Nest** will be set. If it is not nested then the skeleton will run in a flat mode. The nested mode requires few steps to be done before running in parallel. This scheme is general and is used to implement all skeletons. However, there are few variations to the scheme which are specific to the skeleton itself. The steps are:

- Split the original group according to the **split** value.
- Create a new group which consists of all processes whose ranks are zeros in addition to the process with rank 0 in the original group.

[4] EKTRAN is an Arabic word which means a function.

Suppose we have **n** processes allocated for a skeleton, the split argument is **p** and there is another skeleton as the function argument for a skeleton. Simply, the original group is going to be dvided into **p** sub-groups. All processes who have the same **rank mod p** value will form a group and their corresponding rank will be **rank div p**.

The process(es) within each sub-group will be allocated to a skeleton. If the skeleton has another skeleton as its function argument then the above scheme of splitting and creating sub-groups will be repeated to create new sub-groups.

4.1 A Parallel Implementaion of Map

A **map** HOF could be implemented in parallel by using a process farm which is a widely used construct for data parallelism [22]. A farmer process has access to a pool of worker processes, each of which runs the same function. The farmer distributes data to workers and collects results back. The effect is to apply the function on every data item.

However, the above description is for a flat process farm skeleton where no nesting is involved. Therefore, we have generalised the above implementation in order to handle nesting of other skeletons. Figure 2 shows the main algorithm for controlling the process farm skeleton. It handles cases where it could run in a flat mode or in a nested mode. Figure 3 shows the case when the process farm is running in a nested mode and how its workers have to manage nesting.

4.2 A Parallel Implementaion of Fold

A **fold** HOF could be implemented in parallel by using a balanced binary divide and conquer skeleton. Simply, the idea is to apply an argument function across a list in parallel. This could be achieved by using a root node which will divide the original list into a fixed number of sub-lists, send the sub-lists to each intermediate and leaf node in the tree and keep one sub-list for local processing. Next, the root will apply the function on its sub-list and recieve the sub-results from its children.

The leaf nodes in tree will receive sub-lists from root, apply the function on the local sub-lists and then send the sub-results to parents. The intermediate nodes will receive sub-lists from root, apply the function on its local sub-lists, recieve sub-results from children then send the final result to parents.

Figure 4 illustrates how does the divide and conquer skeleton manages nesting. Similar to the parallel implementation of map, we had to generalise the parallel implementaiton of fold in order to handle nesting.

4.3 Nesting Map and Fold

Figures 8, 9, 10 and 11 which could be found in Appendix B show how map and fold skeletons can be nested within themselves and between them as well. The idea is to split the orginal group and assign each nested HOF a sub-group to work on. The splitting of sub-groups will be needed in case there is another level of nesting and so on until no more skeletons appear in the function argument of another skeleton.

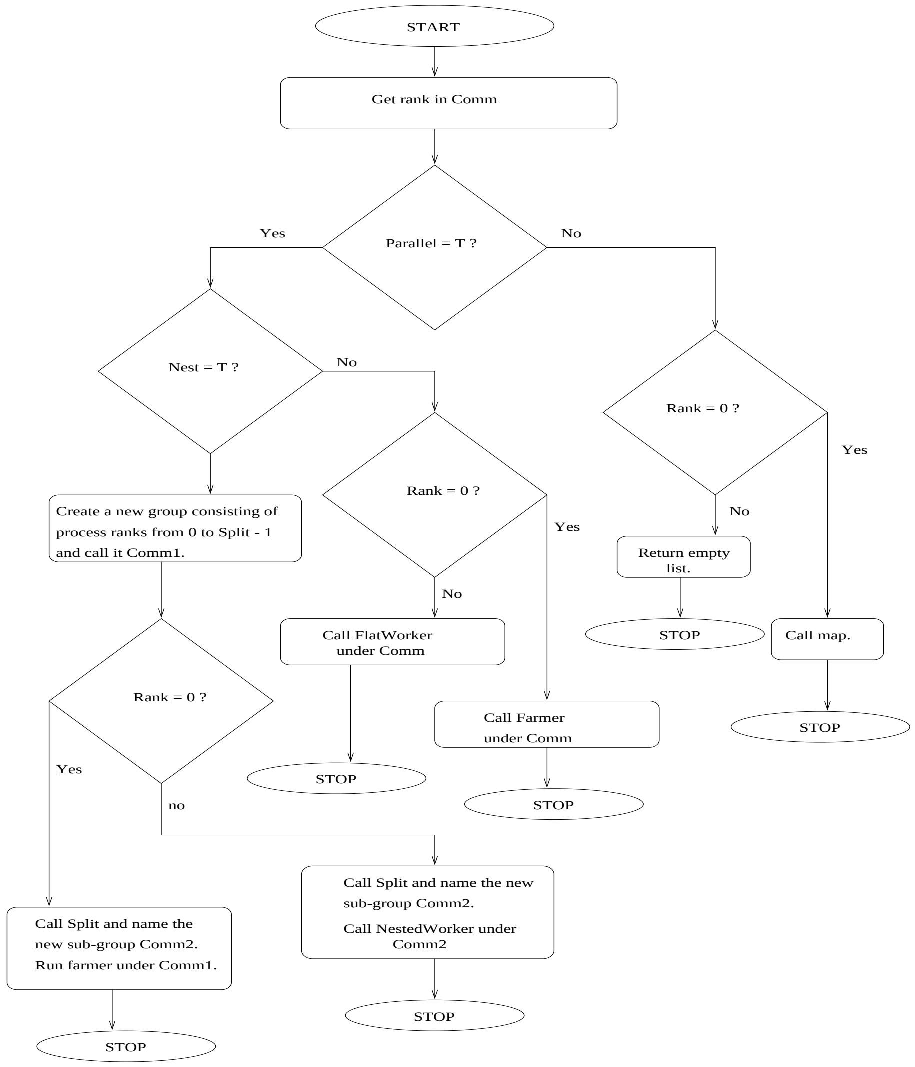

Fig. 2. The main algorithm for controlling the process farm. In the algorithm we use Comm to denote a group as MPI uses communicators to denote groups.

5 Experimental Environment

5.1 The Parallel Compiler

Figure 5 illustrates the various stages involved in compiling an **EKTRAN** application to genetrate machine code to be run on a parallel machine. The main stages of the compilation process are:

- **Front End 1** : in this stage an **EKTRAN** program is scanned, parsed, type checked and transformed into **CAML** [23] source code. All HOFs are converted into parallel skeletons by interfacing the **CAML** program with calls to the C skeleton library.
- **Front End 2** : before feeding the **CAML** program into the **camlot** citeMau95 compiler it has to be manually modified to indicate sites of nested skeletons. This part is to be

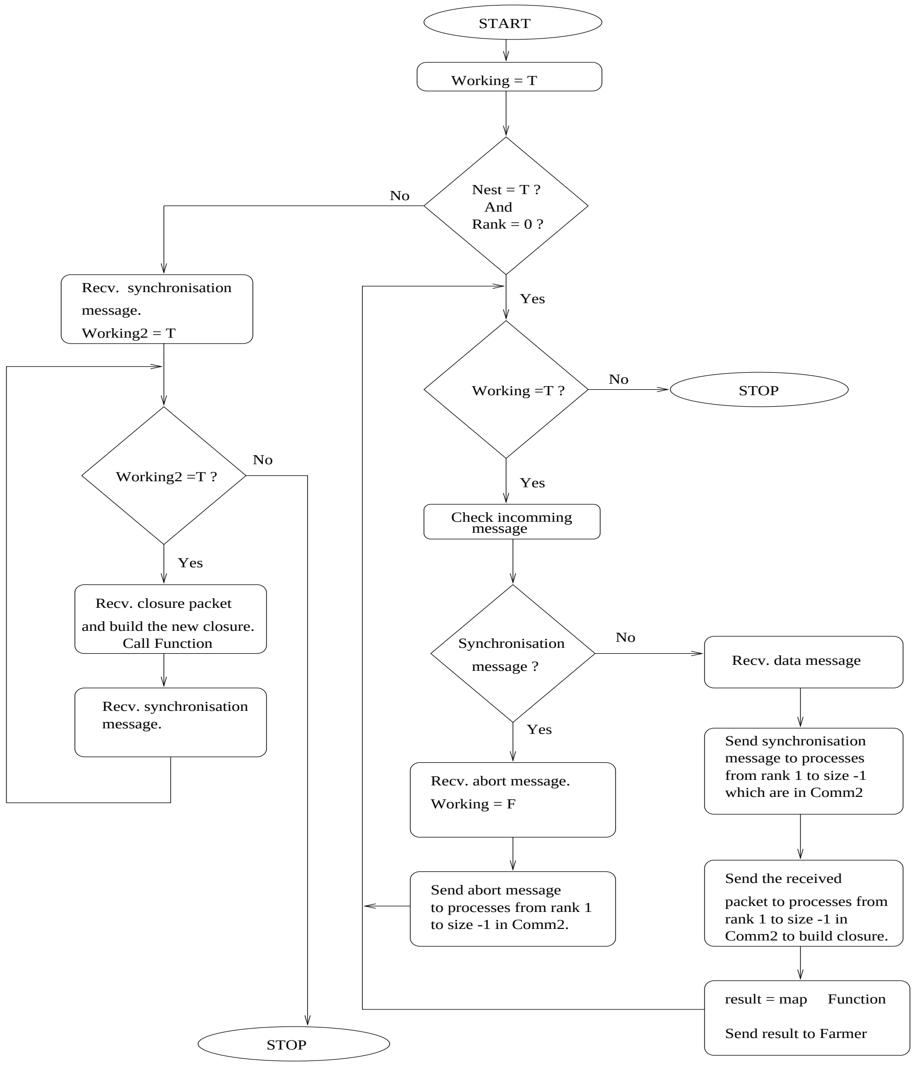

Fig. 3. The algorithm for the nested workers.

automated later on. This stage will generate the equivalent C code for the **CAML** program with all HOFs being converted to calls to the skelton library.

– **Back End** : The **C + MPI** code will be compiled by an ANSI C compiler and the linked with the skeleton library to generate a parallel machine code for the target architecture which is a Fujitsu AP1000 parallel machine at this stage. The advantage of generating **C + MPI** code is the ease of portability which will be investigated later on.

5.2 Test Example - Matrix Multiplication

We have chosen a well-known problem for evaluating our system for nesting skeletons. It is the problem of multiplying two matrices $A_{m\times n}$ and $B_{n\times k}$, resulting in the matrix $C_{m\times k}$. The code for performing the multiplication is shown in Appendix A. In this example, each

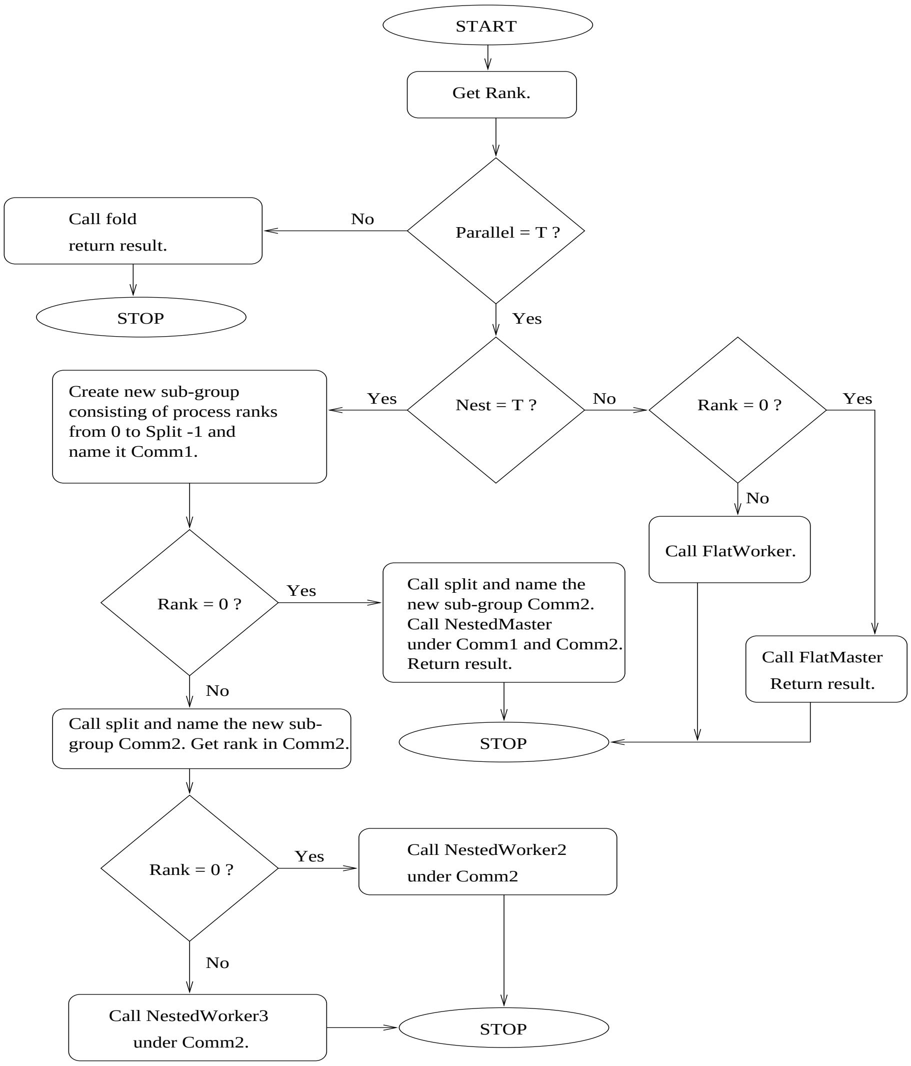

Fig. 4. The main algorithm for controlling the binary divide and conquer.

matrix is represented as list of lists. Furthermore, we have added an extra level of complexity to the problem as we use arithmatic on Arbitrarily Large Numbers (ALN) [24]. Here numbers are represented as lists of decimal digits. Arithmatic is based on mutually recursive functions, ultimately incrementing and decrementing numbers. The recursive functions make the addition or multiplication an expensive operation which depends on the actual values stored in the list. This is needed in order to make the computation costs more expensive than the communication costs as otherwise parallel execution will be longer than sequntial execution.

This test example is a good case for demonstrating nested skeletons up to 3 levels of nesting. The following definition is taken from the Matrix Multiplication program.

```
def sum l = (((fold add) [0]) l);
def MatrixMul m1 m2 =
```

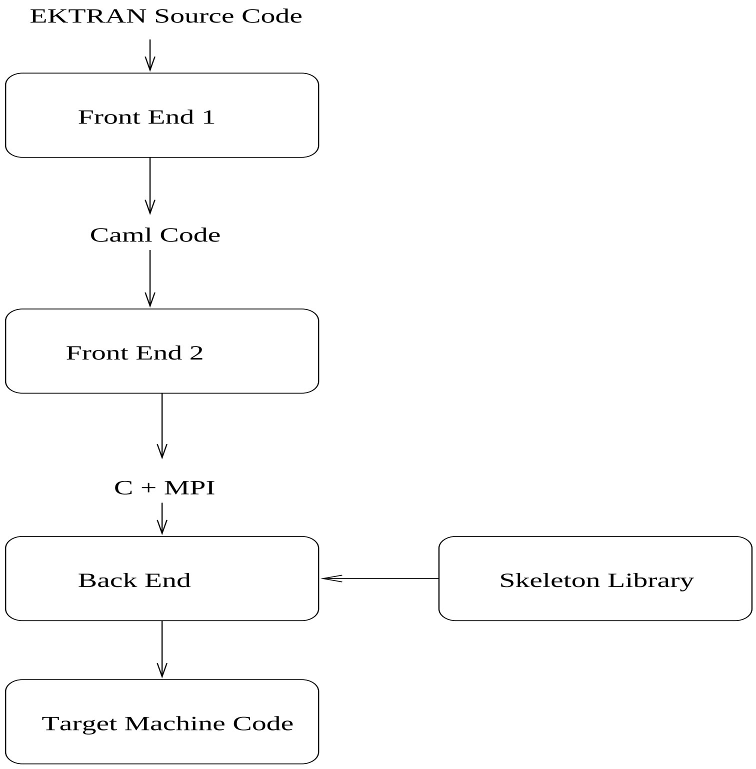

Fig. 5. The compilation environment.

```
((map (map sum)) (((CrossProduct (mmap2 mult)) m1) (transpose m2)));
```

The outer map nests another map (the inner map) which in turn nests a fold which is hidden in the sum function definition. The experimental results for this example are shown in Section 5.3.

5.3 Results

The matrix multiplication program was evaluated on a Fujitsu AP1000 parallel machine. Figure 6 shows the execution time for multiplying two matrices each has 10 rows and 10 columns with numbers consisting of only two digits. The results look slow for such an application because it is expensive to add or multiply numbers by recursive incrementation and decrementation. However, the results show that the scheme we propose for nesting is working. Notice that there is load imbalance because the created sub-groups contain different number of processors which depends on the actual number of processors the application was executed on in the first place. The speedup results for the application with same parameters are shown in Figure 7. The results do not show good speedup results as the application parameters are small and running it in parallel was not of a good advantage over ruuning it sequentially.

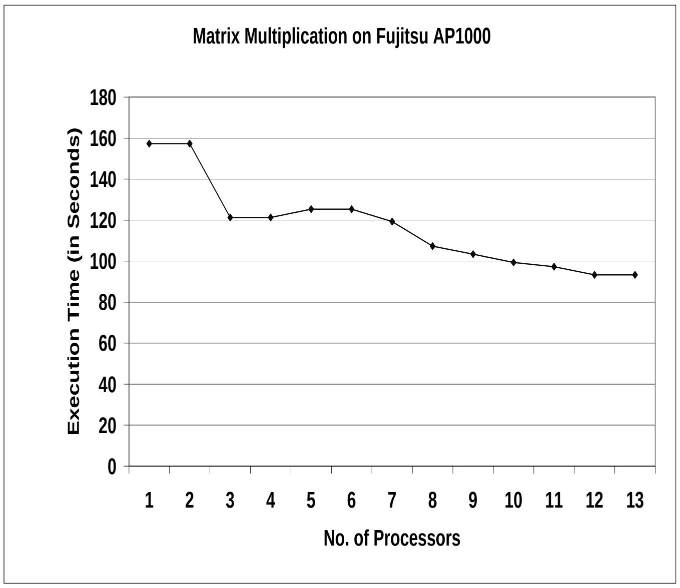

Fig. 6. Matrix Multiplication on Fujitsu AP1000.

6 Conclusions

We have presented a scheme for arbitrary nesting of algorithmic skeletons. The system is semi-automatic at this stage of the project where sites of nested skeletons have to be indicated manually in the transformed code. We intend to automate this part by analysing the program and inserting the needed flags to indicated sites of nested skeletons. Furthermore, the scheme is general enough and it will be used to nest arbitrary skeletons to arbitrary levels.

Early results from the test example do not show good speedup. This is due to the parameters of the problem which are small for a problem to run in parallel. We could not have results for large parameters at this stage as we have problems with large data sets. The problems are due to infinite loops when we try to find the size of large objects before flattening them in order to transmit them over the network.

7 Acknowledgments

We would like to thank the Imperial College Fujitsu Parallel Computing Research Centre for access to their AP1000.

We would also like to thank the British Council and Yarmouk University for supporting this research.

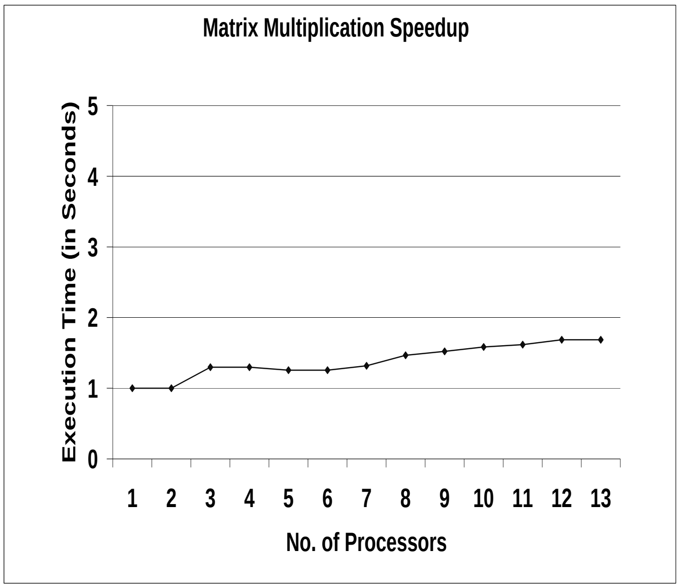

Fig. 7. Matrix Multiplication speedup results on Fujitsu AP1000.

References

1. M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman/MIT, London, 1989.
2. D.B. Skillicorn and D. Talia. Models for parallel computation. Technical report, Computing and Information Science, Queen's University, Kingston Canada, April 1996.
3. T. Bratvold. Determining Useful Parallelism in Higher Order Functions. In *Proceedings of the 4th International Workshop on the Parallel Implementation of Functional Languages, Aachen, Germany*, pages 213–226, September 1992.
4. J. Darlington et al. Structured Parallel Programming. In *Proceedings of Massively Parallel Programming Models Conference*, pages 160–169, Berlin, September 1993. IEEE Computer Society Press.
5. J. Darlington and H. W. To. Building Parallel Applications without Programming. In *Second Workshop on Abstract Machine Models for Highly Parallel Computers*, Leeds, 1993.
6. D. K. G. Campbell. *CLUMPS: a candidate model of efficient, general purpose parallel computation.* PhD thesis, University of Exeter, October 1994.
7. F. A. Rabhi. Exploiting Prallelism in Functional Languages: a Paradigm -Oriented Approach. In *Abstract Machine Models for Highly Parallel Computers*, April 1993.
8. J. Darlington et al. Parallel Programming using Skeleton Functions. In M Reeve A Bode and G Wolf, editors, *PARLE '93 Parallel Architectures and Languages Europe, Munich, Germany*, pages 146–160, Leeds, 1993. Springer-Verlag, LNCS 694.
9. D. Feldcamp et al. Towards a Skeleton-based Parallel Programming Environment. In A. Veronis and Y. Paker, editors, *Transputer Research and Applications*, volume 5, pages 104–115. IOS Press, 1992.
10. D. Feldcamp and A. Wagner. Parsec - A Software Development Environment for Performance Oriented Parallel Programming. In S. Atkins and A. S. Wagner, editors, *Transputer Research and Applications*, volume 6, pages 247–262. IOS Press, 1993.
11. T. Bratvold. *Skeleton-based Parallelisation of Functional Programmes.* PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 1994.

12. D. J. Busvine. *Detecting Parallel Structures in Functional Programs.* PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, April 1993.
13. H. V. Sreekantaswamy. Performance Prediction Modelling of Multicomputers. Technical Report 91-27, University of British Columbia, Vancouver, Canada, November 1991.
14. S. Pelagatti. *A methodology for the development and the support of massively parallel programs.* PhD thesis, Università di Pisa-Genova-Udine, March 1993.
15. M. Daneluto et al. The P^3L Language: An Introduction. Technical Report HPL-PSC-91-29, Hewlet-Packard Laboratories, Pisa Science Center, Pisa, Italy, December 1991.
16. B. Bacci et al. P(3)L - A Structured High-level Prallel Language and its Structured Support. *Concurrency: Practice and Experience*, 7(3):613–641, 1995.
17. R. Rangaswami. *A Cost Analysis for a Higher-order Parallel Programming Model.* PhD thesis, University of Edinburgh, 1995.
18. H. W. To. *Optimising the Parallel Behaviour of Combinations of Program Components.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, London, 1995.
19. J. Backus. Can Programming Be Liberated from the von Neuman Style? A Functional Style and Its Algebra of Programs. *CACM*, 21(8):613–641, August 1978.
20. Ake Wikstrom. *Functional Programming Using Standard ML.* Prentice Hall International, 1987.
21. P. S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, 1997.
22. P. D. Coddington. The Application of Transputer Arrays to Scientific Problems. In John Hulskamp, editor, *Australian Transputer and OCCAM user Group Conference Proceedings*, pages 5–10, Jun 1988.
23. M. Mauny. Functional Programming using Caml Light. Technical report, INRIA, 1995.
24. P. Robinson. From ml to c via modula-3: An approach to teaching programming. In Mark Woodman, editor, *Programming Language Choice*, pages 149–169, UK, 1996. International Thomson Computer Press.

A Matrix Multiplication in EKTRAN

```
def head l = hd l;
def tail l = tl l;
def buildlist x =
if == x 0 then [] else ++ x (buildlist (- x 1));
def makelist f x =
if == x 0 then [] else ++ (f 2) ((makelist f) (- x 1));
def makelistlist f x y =
if == x 0 then [] else ++ ((makelist f) y)
(((makelistlist f) (- x 1)) y);
def pred1 l =
if == l [1] then [] else if == (hd l) 0 then
++ 9 (pred1 (tl l)) else ++ (- (hd l) 1) (tl l);
def pred l =
if == l [1] then [0] else (pred1 l);
def inc l =
if == l [] then [1] else if == (hd l) 9 then
++ 0 (inc (tl l)) else ++ (+ (hd l) 1) (tl l);
def add l1 l2 =
if == l2 [0] then l1 else ((add (inc l1)) (pred l2));
def mult l1 l2 =
if == l2 [0] then [0] else ((add l1) ((mult l1) (pred l2)));
def smap f l =
```

```
if == l [] then [] else ++ (f (hd l)) ((smap f) (tl l));
def transpose l =
if == l [] then [] else if == (hd l) [] then [] else
++ ((smap head) l) (transpose ((smap tail) l));
def mmap2 f xs ys =
if && == xs [] == ys [] then [] else
++ ((f (hd xs)) (hd ys)) (((mmap2 f) tl xs) tl ys);
def sum l = (((fold add) [0]) l);
def CrossProduct f xs ys =
if == xs [] then [] else
++ ((smap (f (hd xs))) ys) (((CrossProduct f) (tl xs)) ys);
def MatrixMul m1 m2 =
((map (map sum)) (((CrossProduct  (mmap2 mult)) m1) (transpose m2)));
((MatrixMul (((makelistlist buildlist) 10) 10))
(((makelistlist buildlist) 10) 10));
```

B Figures for various cases of nesting map and fold

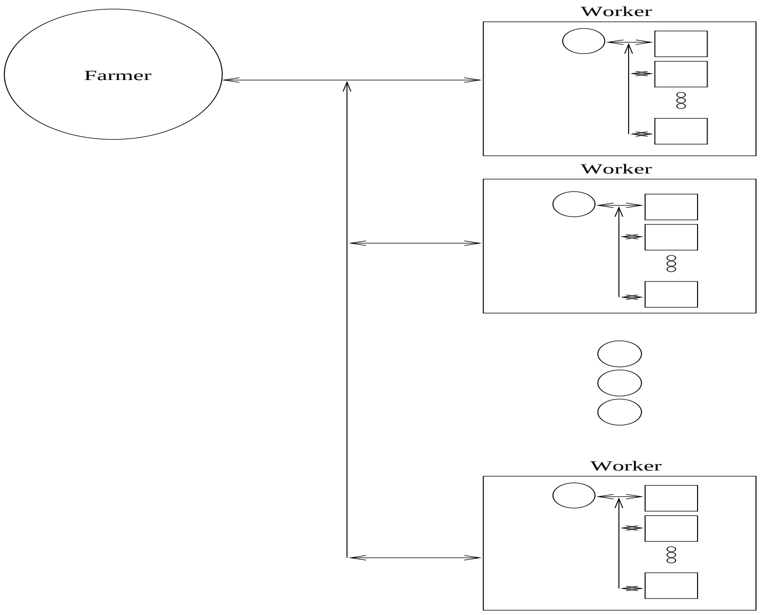

Fig. 8. Nesting map within a map.

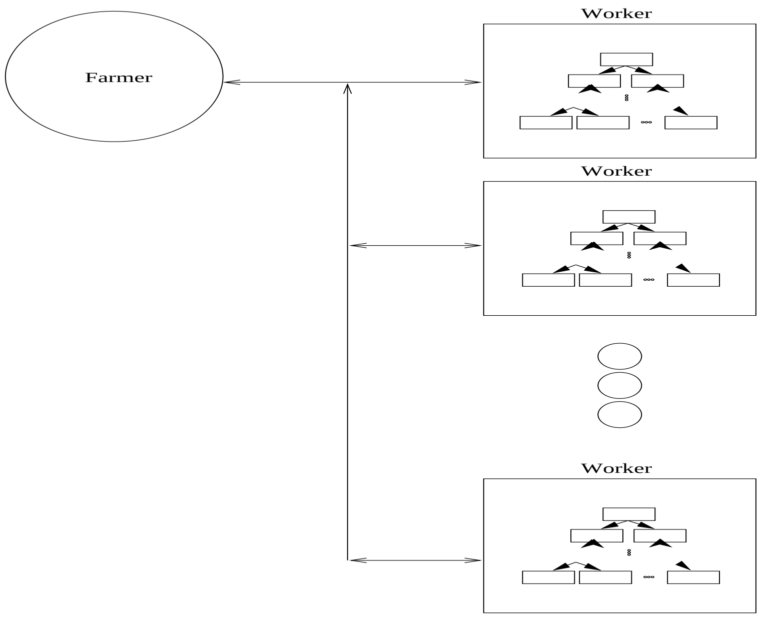

Fig. 9. Nesting fold within a map

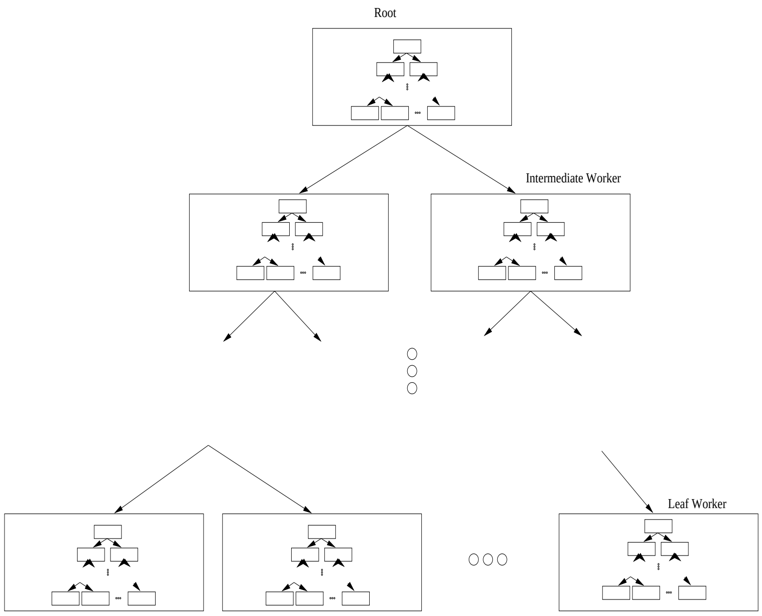

Fig. 10. Nesting fold within a fold.

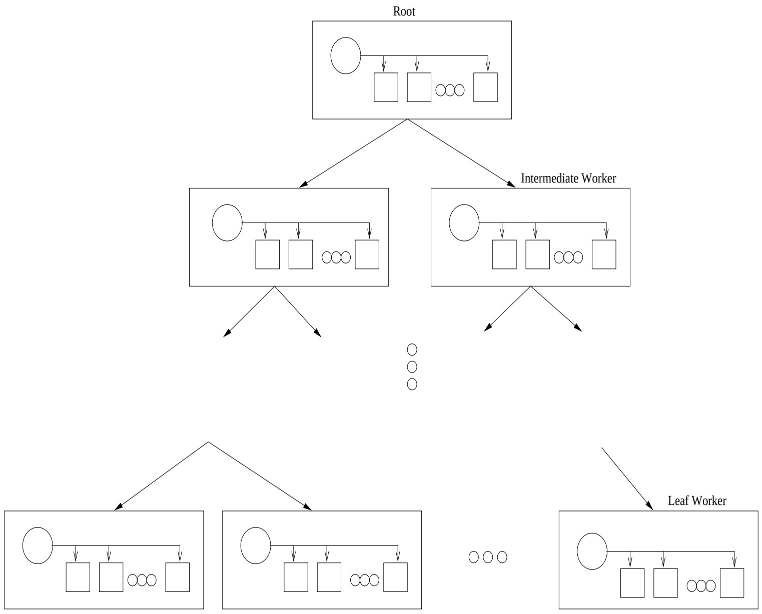

Fig. 11. Nesting map within a fold.