A Framework for Nesting Algorithmic Skeletons

Mohammad Hamdan, Greg Michaelson and Peter King

July 15, 1999

Abstract

We propose a framework for compiling and executing arbitrary depth algorithmic skeleton nesting. It consists of translation rules (schemes) to be used at compile-time for the static-analysis of the source program and at run-time to execute nested skeletons in parallel. We report results for deeply nested programs for different platforms.

1 Introduction

It is well known that parallelism adds an additional level of difficulty to software development. Following Cole's characterisation [2] of algorithmic skeletons, there has been considerable interest in the association between skeletons and higher order functions (HOFs) as a basis for parallel functional programming. However, a major problem lies in the nesting of arbitrary skeletons to arbitrary depth, in particular in automatically generating nested skeleton instantiations from composed HOFs.

Many researchers have worked on skeletons. Pelagatti developed P3L [6] which is an explicit parallel programming language where the programmer has to craft skeletons together. The approach is static in nature as code generation depends on the abstract machine that has been fixed for the construct and on the target architecture at hand. In Rangaswami's HOPP (Higher Order Parallel Programming) model [7] for skeleton-oriented programming, nesting of skeletons was limited to three levels and code had to be generated manually. To [8] investigated optimising the combinations of algorithmic skeletons. A language for combining skeletons was proposed and a set of primitive skeletons was chosen. There have also been a number of approaches based on restrictions to the forms parallelism may take. In Blelloch's NESL [4], parallel operations may only be applied to sequences. In Jay's approach [5], all operations must be shapely, that is they must preserve the physical structure of their operands. Such limitations greatly ease both static analysis of programs and data communication between nested parallel control structures but necessarily constrain the language's parallel expressiveness.

The goal of this work is to allow **arbitrary depth** nesting of skeletons and automatically generate parallel code for the program prototype. Therefore, we propose a framework for compiling and executing arbitrary nested algorithmic skeletons. It consists of translation rules (schemes) used at compile-time for the static-analysis (called preprocessing) of the source program and schemes used at run-time to execute the nested skeletons in parallel.

2 The Programming Model

Our framework is demonstrated through **Ektran** (which means a function in Arabic), a simple functional programming language with polymorphic types which draws on Backus' FP [1]. Ektran provides a number of skeleton-HOFs (called operators) that perform general purpose operations on lists. These operators are inherently parallel and the programmer uses them to express programs in terms of compositions of these sets of operators along with user-defined functions. An Ektran program normally consists of constant bindings, function definitions and an expression to be evaluated. We define a subset of Ektran's grammar below:

```
<prog> ::= {<constant_bindings>} {<function_definitions>} <exp>
<exp>  ::= <value> | <operator> | (<exp> <exp>) |
           <unary_operation> | <binary_operation> |
           if <exp> then <exp> else <exp>
<operator> ::= map | fold | Fold | SFold | @
```

Expressions are the main part of the language and can be a value, an operator, a function application of two expressions enclosed by parentheses, a unary operation, a prefix binary operation or a conditional expression.

3 The Compile-time Framework

3.1 Engineering the Parallel Compiler

The Ektran compiler was engineered by plugging different components together and is organised in three stages: **front end 1**, **front end 2** and **back end**. The purpose of the first stage (Front End 1) is to scan, parse, type-check, preprocess and transform an Ektran program into a CAML Light [3] source program with calls to the skeleton library using CAML Light's C interface. The next stage (Front End 2) generates a C program from the CAML Light source program using the Camlot (which is a high performance CAML Light to C) compiler. Finally, target code can be generated for different target architectures during the last stage (Back End) using an ANSI C compiler. The current compiler supports only the CRAY T3D, the Fujitsu AP1000 and a network of Linux workstations. The skeleton library is linked at compile-time to the generated code.

3.2 Preprocessing

To preprocess an Ektran program a number of schemes are applied to generate a nesting structure which can be used to combine the corresponding skeletons. The schemes are applied to Ektran's syntactic objects to produce equivalent syntactic objects as a result in the following order:

- **SymbolicBetaReduction**, lifts all HOF combinations to the top level of the program. To do so, Ektran expressions are partially beta reduced (i.e. only symbolic beta reduction is applied) by replacing all functions appearing in the main expression by their bodies. This greatly eases the **nesting deduction** stage.
- **PipelineNormalisation**, extracts pipeline parallelism which may arise from composed functions. This involves two steps: 1) normalising the composed functions by generating a list of functions and 2) flattening the generated list by deleting multiple occurrences of function compositions.
- **NestingDeduction**, annotates the skeleton-HOFs in the abstract syntax tree and generates a nesting structure which is used to combine the corresponding skeletons. For example, suppose HOF_1 has HOF_2 nested, then the scheme will simply annotate the first HOF in the following way HOF_1^N where N means that the function argument is a HOF. The annotations are used by the translator to set the proper flags for the skeleton's C interface. Due to space limitation we outline in Figure 1 only the basic rules for this scheme in an ML-style notation.
- **FreeVariablesDetection**, annotates the outer skeleton-HOF in case free variables are found in the inner skeleton-HOF. The free variables will be transmitted at run-time to guarantee that closures being built are consistent across all processors.

4 The Run-time Framework

4.1 Supported Skeletons

The current supported parallel operators are the **map**, **fold**, **Fold**, **SFold** and **compose** HOFs with equivalent algorithmic skeletons realised as MPI process groups. Parallel communication between processes is effected through MPI and generic Camlot functions for flattening and unflattening data structures. Below we explain in more detail each operator and its corresponding parallel implementation:

- **map**, which applies a function to each element of a list, is realised as a broadcast process-farm skeleton. The process farm is a widely used construct for data parallelism. The farmer process has access to a pool

NestingDeduction $[\![$ constant_bindings | defs | exp $]\!]$ =
$[\![$ constant_bindings | defs | **Deduce** exp $]\!]$

Deduce $[\![$ (exp1 exp2) $]\!]$ = if **LookFold** exp1
then if (**IsHOF** exp2)
then $[\![$ ((**AnnotateHOF** exp1) (**Deduce** exp2)) $]\!]$
else $[\![$ ((**Deduce** exp1) (**Deduce** exp2)) $]\!]$
else if (**IsHOF** exp1)
then if (**IsHOF** exp2)
then $[\![$ ((**AnnotateHOF** exp1) (**Deduce** exp2)) $]\!]$
else $[\![$ ((**Deduce** exp1) (**Deduce** exp2)) $]\!]$
else $[\![$ ((**Deduce** exp1) (**Deduce** exp2)) $]\!]$
Deduce $[\![$ (pipeline l) $]\!]$ = $[\![$ **DeducePipeline** (pipeline l) $]\!]$

Figure 1: The nesting deduction scheme.

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.

- **fold**, which applies a function between each element of a list, is realised as a broadcast binary divide and conquer skeleton. The top level node (root) divides the original list into a fixed number of sub-lists and broadcasts the sub-lists to all nodes (which includes intermediate and leaf nodes) below it in the tree and keeps one sub-list for local processing. Next, the root will apply the function on its sub-list and receive the sub-results from its children. The leaf nodes in the tree will receive sub-lists from root, apply the function on the local sub-lists and then send sub-results to parents. The intermediate nodes will receive sub-lists from root, apply the function on its local sub-lists, receive sub-results from children then send the final result to parents.
- **Fold**, is a special case of **fold** where the function argument is of the form **fn** h => **fn** t => $f1\ (f2\ h)\ t$.
- **SFold** is a special case of **Fold**. In case the function argument for Fold is not associative it might be possible to replace it with two functions where one of them is associative. Its parallel implementation is similar to Fold's implementation.
- **compose**, where the result of one function is the argument for the next, is realised as a pipeline. It is usual to express an arbitrary depth composition as folding binary compose over a list of functions, but this restricts the composed functions to having the same domain and range type. In contrast, we have implemented an unrestricted compose,

where the domain type of each function must be the same as the range type of its argument function. However, the compose pipeline will only work effectively when nested within another skeleton which generates a sequence of data values.

4.2 Skeleton Nesting

The run-time schemes for arbitrary depth nesting of algorithmic skeletons have been designed and implemented as part of the scheduler for each parallel implementation of the supported skeletons. They were built on top of MPI's routines for managing message-passing groups which has the advantage of making our system dynamic rather than static. They allow skeletons to be nested within themselves and within other skeletons. The basic idea is to allocate processes to skeleton groups top-down through traversal of the HOF nesting structure. If a skeleton nests other skeletons (i.e its function argument is another skeleton) then it reallocates processes in its group in order to support the nested skeletons. This involves dividing the group into sub-groups for each of its immediately nested skeletons, and creating a top group which connects all sub-groups. The process then continues recursively.

We define the **SplitGroup** scheme below. The scheme divides or splits a given group (which is a set of processors) into a fixed number of sub-groups. It takes two parameters. The first is the original group and the second is the split value or key which determines how many sub-groups to be created out of the original group.

```
SplitGroup OriginalGroup Key =
let size = FindSize of OriginalGroup
Create key sub-groups
SubGroup_i = {}  where 1 <= i <= key
for i= 0 to (size - 1)
  add i to the (i mod key)th Sub-Group
return  Sub-Groups
```

The **CreateTopCommGroup** scheme is also defined. The scheme creates a new group which consists of the first element in each sub-group that was created by the **SplitGroup** scheme. It takes two parameters. The first is the original group which was divided by the **SplitGroup** scheme, and the second is the same split value (key) used for the **SplitGroup** scheme.

```
CreateTopCommGroup OriginalGroup  Key =
let size = FindSize of OriginalGroup
let CommGroup = {}
for i=0 to (size - 1) step Key
  add i to CommGroup
return CommGroup
```

Figure 2 gives a simple visualisation of how a group can be divided into sub-groups and how a new group gets created from the existing sub-groups by applying the above schemes to a group.

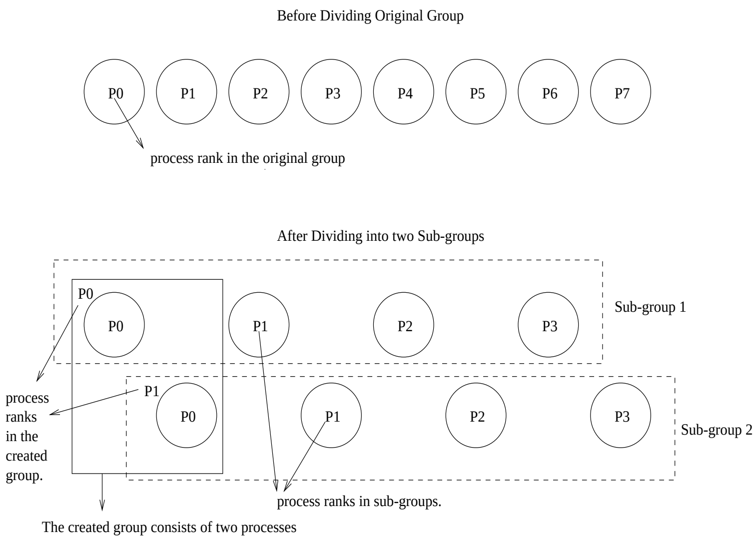

Figure 2: Dividing and creating groups from an existing group.

5 Evaluation

5.1 Test Example I: Matrix Multiplication

Figure 3 shows the execution time and speedup results for an example of 3 levels of **map** nesting in solving matrix multiplication of 40-digit arbitrary length numbers for the Cray T3D, Fujitsu AP1000 and a network of Linux workstations. For this example we exploit parallelism for two cases: 1) running in parallel all nested HOFs (denoted by 3P on the graph) and 2) executing in parallel only the first two nested HOFs (2P). Note that running in parallel only the first two nested HOFs give better results compared to three HOFs. The reason is that the bottom HOF should run sequentially as running it in parallel would only waste resources. Also, note how speedup stays the same for a range of processors then suddenly improves. This has to do with groups being balanced as more processors are added.

5.2 Test Example II: Merge Sort using a Pipeline

Figure 3 shows the execution time and speedup results for a merge sort of 50-digit arbitrary length numbers on a Fujitsu AP1000. In this example, We use a heterogeneous pipeline which consists of two stages: the first uses a **map** HOF and the second uses a **fold** HOF. The novelty here is in running the inner stages of the pipeline in parallel as well. However, the results are disappointing as the best speedup was 2.11 on 7 processors. Nontheless,

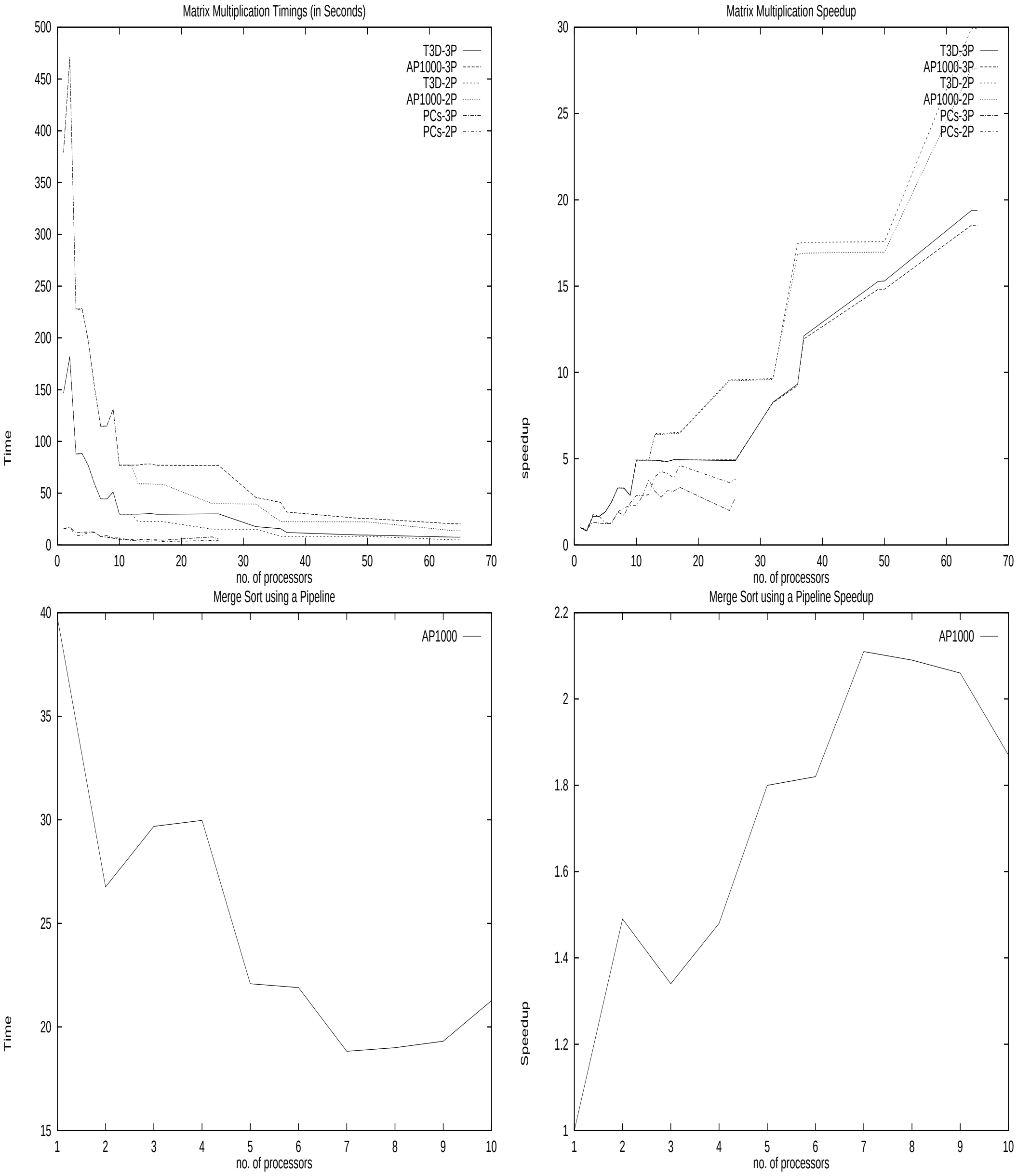

Figure 3: Matrix multiplication and merge sort execution times and speedups

this speedup proves that running the inner stages in parallel improves the performance as speedup for 2 processors was 1.49. We think there are three reasons for the poor performance: 1) the parallelism exploited in the inner stages of the pipeline is poor 2) there are only two stages in the pipeline for this example and 3) it is possible that the pipeline is not load-balanced (i.e one stage needs more or less processors than other stages).

6 Conclusions and Future Work

We have outlined a framework for arbitrary depth skeleton nesting in which nesting is handled at run-time and the code generated does not depend on the target architecture. The results for the matrix multiplication suggest that good cross-platform portability and behavioural consistency can be achieved. However, the results for merge sort using a pipeline suggest that a better example is needed to study the behaviour of the heterogeneous pipeline. The current Ektran implementation is fully automatic. However, the run-time system uses a simple heuristic (the square root of group size) to divide the process groups. The method used to divide groups affects skeleton performance as it controls the number of processes allocated to each skeleton. This is very important in order to load-balance the pipeline. Therefore, we are developing a process allocation scheme, based on sequential Ektran program profiling, cost models for skeletons and performance characterisation of target architectures. The allocation scheme will decide how to divide the groups to achieve the best performance.

References

[1] 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.

[2] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman/MIT, London, 1989.

[3] G. Cousineau and M. Mauny. *The Functional Approach to Programming.* Cambridge University Press, Cambridge, UK, 1998.

[4] G. Blelloch et al. Implementation of Portable Nested Data-Parallel Language. In *Proceedings of the 4th ACM Conference (PPoPP)*, pages 102–111, San Deigo, CA, USA, 1993.

[5] C. B. Jay. Shape Analysis for Parallel Computing. In *Parallel Computing Workshop: Imperial College*, September 1995.

[6] S. Pelagatti. *Structured Development of Parallel Programs.* Taylor and Francis, London, 1998.

[7] R. Rangaswami. *A Cost Analysis for a Higher-order Parallel Programming Model.* PhD thesis, University of Edinburgh, 1995.

[8] 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.