Chapter 1

INTRODUCTION

The super computer of yesterday has become outdated today and the same of today may be obsolete tomorrow. The technology is developing, size of computers are decreasing but the power of computation is increasing enormously. The apparent key to this ever increasing thirst for high speed computation is simultaneity i.e. doing computation in parallel. The parallelism can be incorporated both at organisation level (Hardware) and at programming level (Software).

At the organisational level, to exploit parallelism, considerable progress has been made in the direction of constructing highly parallel machines. These machines avoid the classic ‘Von Neumann bottle neck’ (6) by being effectively decentralised and they are also extensible and practically feasible. The basis of most high-performance computer systems is Multiple Instructions stream and Multiple Data stream (MIMD) organisation. These systems employ multiple processors which execute independent instruction streams accessing data autonomously. The design of such systems is based on the shared memory concept. These models produce severe memory contention problems when processors try to access data residing on the same memory module. To overcome this problem the distributed memory MIMD model was introduced which is scalable to higher orders of parallelism.
Chapter 1. INTRODUCTION

Though degree of parallelism is less another organisation model with parallelism is single Instruction stream and Multiple Data stream (SIMD). In this different processing elements are tied together under the control of one control unit. These P.E.s perform the same function on different data in a locked up step fashion. This is very much suitable for vector-processing techniques.

At the software level, the development in parallel computing in hampered by the bare fact that it is very difficult to implement languages on parallel machines. The reasons being:

- The programmer has to devise a parallel algorithm to meet the requirements.
- The algorithm has to be identified in the form of sequential activities, called tasks or processes, which will run concurrently.
- The data shared between tasks has to be protected by the programmer to prevent from hazards.
- Some times (as in OCCAM on transputers), the programmer is also responsible for mapping each task onto a processor and ensuring that it is physically connected to all other task running processors'.

So, the conventional languages are difficult to program in a parallel environment. An alternative to the imperative languages is the applicative model of computation based on applicative languages also know as functional languages.
Chapter 1. INTRODUCTION

1.1 Parallel Processing: Functional language approach:

During the last few years there has been a growing interest in a new class of programming languages called declarative languages which are naturally compatible with parallel processing and do not depend on the programmer to specify parallelism. These languages allow the programmer to think declaratively specifying what is to be done rather than imperatively specifying precisely how a task is to be done.

Functional languages, which form a sub-group of the declarative group, have a mathematical base and possess the useful property of 'referential transparency' and thus lend themselves naturally to a parallel mode of evaluation. The characteristics and advantages of functional programming are:

1. Functional program contain no assignment statement, so variables once given a value never change i.e., a data item can not be used as both an operand and result within the same program instruction.

2. Functional programs contain no side effects of any kind.

3. A functional call can have no effect other than to compute its result. This makes the order of execution irrelevant since no side effects can change the value of an expression, it can be evaluated at any time. Hence, the burden of presenting the flow of control is not there.
Chapter 1. INTRODUCTION

4. Since expression can be evaluated at any time, one can freely replace variables by their values and vice-versa i.e., programs are referentially transparent. Therefore, the programs are tractable mathematically.

5. A functional language would increase productivity because the algorithm is expressed more concisely in a functional language than in an imperative one. Some more reasons which make the functional program suitable for parallel programs are (40)

- Functional languages are more problem oriented than conventional languages and thus the conversion from a formal specification to a functional program is much shorter and easier.

- They have simple mathematical basis, the $\lambda$-calculus and because of the lack of side-effects, program correctness proofs are easier.

- Functional programs are generally shorter than thick conventional counterparts and thus easier to enhance and maintain.

- Functional languages seem to provide an answer to the problem of exploiting parallelism offered by multiprocessor systems.

A program in a functional language consists of functions, declared through definitions and an expression for evaluation. Application of a function to its arguments is basic operation here (hence they are also known as applicative languages). The arguments need not necessarily be data items, rather they also can be functions, leading to the concept of
Chapter 1. INTRODUCTION

higher-order functions. Similarly an application can return a function besides a value on
its result. Expressions in these languages have natural representation as tree structures (or
graphs if sharing of subexpression is allowed). They are evaluated through graph
transformation steps called reductions which can be done in parallel.

In the reduction phase, main issues involved are reduction order and argument passing
mechanism. Reduction may proceed in a variety of orders but normal order and
applicative order are mainly used [10]. The normal order strategy selection the top left-
most (outer-most) redex for execution while applicative order selects all inner-most
redexes. Any reduction order, if it termination, will lead to the same result. Normal order
is absolutely safe i.e. it is bound to terminate unless the problem itself is non-terminating.
The applicative order is unsafe but it allows better exploitation of parallelism in a program.

An important aspect of, reduction order is ‘laziness or non-strict evaluation’ i.e. performing a computation only when needed. Normal order is inherently lazy while
applicative order is ‘eager’ to compute every thing irrespective of the need for it.

Implementation of non-strict functional languages on parallel architectures is the theme of
the present thesis.

1.2 Motivation:

The change of scene from conventional languages and machines to declarative
languages and data flow/reduction machines is on the way but it may take a long time, in
the mean time we are stuck with conventional von-Neumann machines. So to make use of
existing machines, recent developments [6,28] have focused on efficient computations
Chapter 1. **INTRODUCTION**

of functional languages on conventional Von-Neumann processors. In this direction several abstract machines have been reported: TIM, G-Machines, Spinless-Tagless G-machines [17] etc. Computation of functional languages for sequential processor can be extended to accommodate parallel machines constructed from the Von-Neumann processors in a straight forward way. To date however, little attention has been devoted to develop compilation strategies for vector array processors. Most super computers employ vector registers to gain high performance, exploiting data parallelism while retaining a single flow of control. Single instruction multiple data architectures encompass both array and vector processors. The work (This thesis) is in the area of developing compilation strategies for functional languages on SIMD architectures.

The goal of this thesis is concerned with incorporating data parallelism into a non-strict functional language and concentrates on the effects a non-strict semantics has on data-parallelism. The navel impact of this work on data-parallel programming is that non-strictness enables the control of where parallelism occurs, to be decoupled from a parallel algorithm. Hughes [39] observed that non-strictness gives us new ways in which program can be modularised into parts, and subsequently, the sub-parts “glued” back together. By using non strict evaluation, the two functions can be combined in such a way that only these elements which are actually required are generated Von-strict evaluation therefore provides a special kind of “glue” for combining sub-problems.
Chapter 1. INTRODUCTION

Adding non-strictness to a data-parallel language provides new and interesting ways to modularize parallel algorithms. By using lazy evaluation techniques, it is possible to couple the control of where evaluation occurs within a data-parallel object from an algorithm that manipulates the object. From a slightly different perspective, the work in this thesis could also be used to cure some of the gross inefficiencies of standard non-strict functional languages by improving their implementation of arrays.

1.3 Parallelism:

As a single instruction perform operations on a single word of a machine's memory, so a monolithic data-parallel operation operates on collections of words in parallel. Such a collection of words constitutes a data-parallel object in which each word can be interpreted as the contents of one of the processes of a data-parallel machine.

The essence of data parallelism, in this work, is abstracted to form the map function. Instead of converting collections of simple operations such as addition to form a monolithic vector operations such as addition to form a monolithic vector operation such as vector-addition, map applies an arbitrarily complex function synchronously and in parallel to each element of a monolithic data-parallel object. Using map all the transformations occur to a single element of a parallel object are considered and these transformations are modelled by a single function that is applied to each element of the parallel object.
Chapter 1. INTRODUCTION

The second observation is encapsulated by the fold function and can be thought of as “fitting” as associative operator between the successive elements of a parallel object to create a large expression which is evaluated to a single value, fold can be considered as modelling a general reduction that would otherwise be expressed as an iterative construct in an imperative language.

1.4 Organisation of the thesis:

This thesis is organised as follows:

- Chapter 2: Review of functional programming: In this chapter a brief introduction to functional programs, lambda calculus and parallel functional programming are given as a prelude to implementation of functional languages on parallel Architectures.
- Chapter 3: Laziness and Parallel Data Structures: In this chapter a discussion is made on how non-strictness and data-parallelism don’t quite fit together and propose a new evaluation mechanism termed the “aim of evaluation” to remedy the problem. This chapter also describes data-parallel extensions incorporated into the lazy functional language. PODs, parallel data structures that share many of the characteristics of Haskell arrays are described – their distinguishing feature is that they are unbounded. POD comprehensions, a framework within which communication and parallel operations on PODS can be expressed, are also presented. The semantics of these extensions are given in terms of translation rules into a core set of primitive parallel operations.
Chapter 1. INTRODUCTION

- Chapter 4: Parallel Algorithms: This chapter gives the definition of a set of fundamental higher-order functions that encapsulate general patterns of data-parallel computation. The parallel algorithms developed will form the building blocks for many parallel applications and express the use of non-strictness in data parallel algorithms.

- Chapter 5: Algorithm for SIMD architectures: This describes the program transformations that "Vectorize" an enriched lambda calculus that has been extended with the primitive parallel operations of chapter 3 and Chapter 4. Of particular interest in the way in which the algebraic data types and higher-order functions are vectorized so that the algorithms are very much suitable for SIMD architectured machines.

- Chapter 6: PFPM-An abstract Machine: A general Functional programming machine is considered and a special Intermediate code is developed so that it works as an abstract machine to implement the required functional program on SIMD machines. Built in functions, transformation rules etc. are also described.

- Conclusion, technical contribution and further work are given lastly.