Chapter 1

Introduction

The two aims, on the one hand for highly-parallel hardware, and on the other for easy and speedy creation of high-quality software, are seen by many to be directly antithetic. J.P. Eckert wrote, when arguing for parallel data transfer and arithmetic in computers of EDVAC's generation, that

> The arguments for parallel operation are only valid provided one applies them to the steps which the built in or wired in programming of the machine operates. Any steps which are controlled by the operator, who sets up the machine, should be set up only in a serial fashion. It has been shown over and over again that any departure from this procedure results in a system which is far too complicated to use [Eck46].

The quest to overturn this wisdom, which had been learned "over and over again" in 1946, has occupied a large portion of the computer science community since then. Why is parallel programming difficult?

- **Performance:** The performance of a parallel program is difficult to optimise—counting the number of instructions is no longer good enough, because some of the instructions may be executed simultaneously.
- **Portability:** There are many more ways in which two parallel computers may differ, and these can mean that quite different algorithms are suitable for different target architectures.
- **Determinacy:** The order of events during parallel program execution is almost always indeterminate. The program's output is determinate only if it is written carefully.

All of these problems do arise to some extent when programming sequential computers, but in the general case of parallel computing they are epidemic.

1.1 Functional programming

The main subject of this book is the interesting and powerful class of functional programming languages. The reason for choosing such a language is the ease with which

such programs can be manipulated algebraically, and the bulk of the book is devoted to introducing and demonstrating this in action.

It is through algebraic manipulation of programs that the problems of parallel programming are addressed. We retreat from the hope that a single program will serve for all the different parallel computers we might wish to use, and instead begin with a single specifying program. Versions for different target architectures can then be derived by the application of a toolbox of mathematical transformations to the specification, leading to versions tuned to the various machine structures available. The transformation pathways can then be re-used when modifications to the specification are made.

1.2 Loosely-coupled multiprocessors

Parallel programming is much simplified if we can assume that interprocessor communication is very efficient, as in a shared memory multiprocessor. This book is about programming a much larger class of computers for which such simplifying assumptions do not hold. In general, there are two distinct problems in mapping a parallel program onto a computer: *partitioning* and *mapping*. The most important simplifying assumption often made is to avoid mapping, and assume that performance is independent of where processes are placed. The class of loosely-coupled multiprocessors is defined to characterise architectures where this assumption is not valid: a loosely-coupled multiprocessor is a collection of processing elements (PEs), linked by an interconnection network with the property that communication between "neighbouring" PEs is much more efficient than communication between non-neighbours. Depending on the interconnection network's topology, there are many varieties of such an interconnection network. The important feature is that not all PEs are local to one another, so that process placement is important to program performance.

The importance of this class of architectures is that they are easy and inexpensive to build on a large scale. It is not, therefore, surprising to find quite a number of loosely-coupled multiprocessors on the market and in use. Examples include Meiko's *Computing Surface*, Parsys's *Supernode* and Intel's *iPSC*.

In architectures of this kind the full generality of the software design problems for parallel computers become apparent. We find that data communication is often a primary computational resource, and that much of the algorithm design effort is aimed at reducing a program's communications demands. Several examples are given of how this can be done using program transformation. The techniques have application to other parallel architectures including more closely-coupled machines and SIMD computers.

1.3 Neighbour-coupled multiprocessors

A neighbour-coupled multiprocessor is a more idealised abstract computer architecture, and is introduced here as an experiment. A neighbour-coupled multiprocessor is a loosely-coupled multiprocessor, where each PE is very closely coupled to its neighbours, so closely that the programmer can assume that a PE can read and write its neighbour's memory as quickly as its own.

We shall return to this abstract architecture later in the book to examine whether it allows useful simplifications.

1.4 A reader's guide

The book consists of the following components:

- **Chapter 2. Functional Programming:** This chapter introduces functional programming from first principles. The programming language is presented by means of examples. Simple techniques are given for manipulating programs to modify their structure while retaining the same input/output mapping. These are augmented by a handful of induction rules for proving generic properties about programs.

  The language is based on Miranda[1] and Haskell (a public-domain language design for which a specification is in preparation [HWA+88]).

- **Chapter 3. Sequential and Parallel Implementation Techniques:** The aim of this chapter to sketch how our functional language might be compiled to run efficiently on a conventional computer, and to examine how this scheme (graph reduction) might be extended for a tightly-coupled multiprocessor.

- **Chapter 4. Specifying and Deriving Parallel Algorithms:** This chapter examines how parallelism and inter-process communication are manifest in a functional program script. Horizontal and vertical parallelism are identified and examples are given in the form of divide-and-conquer and pipeline algorithms respectively. The main emphasis in this chapter is the development of program transformation techniques. Examples are given of introducing pipeline parallelism, and of transforming a divide-and-conquer algorithm into a cyclic "process network" program. This is illustrated by application to a simple ray tracing program.

- **Chapter 5. Distributed Parallel Functional Programming:** We can write programs for which a good placement onto a loosely-coupled multiprocessor can be made. This chapter applies a declarative programming language approach to actually specifying this placement. It incorporates abstraction mechanisms to give concise mappings for regular architectures and algorithms. The notation is illustrated with several examples.

- **Appendix A. Proofs and Derivations:** This appendix gives proofs and derivations which would have cluttered the presentation given in chapter 4. Although quite dense later on, the earlier material in this chapter is quite tutorial in nature and might be read concurrently with Chapter 4 by those more interested in program derivation and verification than in parallel programming.

- **Appendix B. Common Definitions:** This appendix lists widely-used function definitions for easy reference.

[1] Miranda is a trademark of Research Software Ltd.

- **Appendix C. Programming in a real functional language:** The functional language used in this book is not quite compatible with any commonly-available language implementation. This appendix lists the small (and quite innocuous) differences from Miranda in order to aid a reader who wishes to experiment.

Each chapter ends with some pointers for the interested reader towards other books, articles and research papers which might be of interest.