Chapter 1

Parallel programming

In fact the advent of parallel programming may do something to revive the pioneering spirit in programming, which seems to be degenerating into a rather dull and routine occupation

S. Gill. Parallel programming. Computer Journal vol. 1, 1958.

§1 Introduction

This thesis introduces a new programming paradigm called *definitive* (definition-based) programming, and examines its applicability to both parallel programming and the modelling of concurrent systems. The advantages that definitive programming offers are highlighted, and illustrated by worked examples.

We have developed a computational model for synchronous execution of definitive programs, and a specification notation for modelling, which can be transformed into an executable simulation. We use techniques that are both cognitively and theoretically based, which in the case of the modelling of concurrent systems has the advantage of permitting both intuitive reasoning and the potential for formal analysis of behaviour. The central thesis is that definitive programming has a contribution to make to the overall goal of generating, understanding and verifying parallel programs.

§1.1 Thesis outline

This chapter considers how difficult it is to construct working programs, both sequential and parallel. An outline is given of existing approaches to programming, with particular reference to their applicability to parallel programming. This examination highlights certain issues, such as data dependencies, which are not effectively addressed using current techniques. The purpose of this chapter is to motivate the need for, and development of, the definitive programming paradigm which is described in the rest of the thesis - it is not required for understanding definitive programming.

Chapter 2 introduces definitive programming by a description in terms of state-transition models of computation. An abstract definitive machine is outlined and used to demonstrate how definitive programming can be used for parallel programming. Chapter 3 addresses the technical issues of implementing a simulator for the abstract definitive machine.

Chapter 4 gives an example of developing a program for the adm, and provides a proof that execution of the program cannot lead to interference. It also indicates some deficiencies in the abstract definitive machine, as a motivation for Chapter 5.

Chapter 5 describes a method for modelling and simulating concurrent systems. It introduces a specification notation called LSD, which is used to describe the behaviour of a system in an agent-oriented manner. We show how to transform an LSD specification into a family of abstract definitive machine programs. A transformed program can then be executed by the abstract definitive machine, to allow the behaviour described by the LSD specification to be simulated.

Chapter 6 gives an LSD specification of a telephone system, and then transforms the specification using the techniques of Chapter 5 to produce a family of adm programs. Chapter 7 develops a program for use in an educational environment, by modelling a system using an LSD specification and then writing an executable adm program.

Chapter 8 draws together the themes of the thesis. It contains an appraisal of the use of the abstract definitive machine and the LSD specification notation, a comparison between definitive programming and other paradigms, and a description of possible extensions and improvements to the adm and our method for modelling and simulation.

Chapter 9 summarises the contribution of the thesis, both in terms of the advantages offered by definitive programming and a description of the issues which have not been resolved. It concludes by describing what would be the most useful future research in the area.

Appendix 1 describes existing methods for describing concurrent behaviour. Appendix 2 describes what is meant by the term reactive system. Appendix 3 contains the user documentation for `am`, the abstract definitive machine simulator, and Appendix 4 the maintenance documentation. Appendices 5 to 8 contain testing of various example contained in the body of the thesis. Appendix 9 contains a listing of the source code for `am`. Appendix 10 gives a glossary of terms which are used with a special meaning in the thesis.

§1.2 Programming is difficult

Our ability to construct large working programs has always lagged behind our capacity to envisage them. To demonstrate this we give a brief historical overview of the development of programming. When general purpose computer hardware began to appear in the 1950s, attention was focussed on the problems involved in getting these to work correctly [Dijkstra 88]. This led to many technological innovations in the physical equipment used, such as transistors, real-time interrupts, parity checks, and so on. It is probably true to say that programming skills were relegated in importance, since it was felt that more machine access and faster execution would ease the burden of coding and debugging programs for large systems.

The problem of software development was, however, made very apparent in the 1960s, when correctly functioning complex general-purpose hardware systems were becoming available. The imbalance between the achievements in software and hardware technology culminated in a realisation that the computer industry was in what was termed a *software crisis*, a widespread inability to produce large working systems within the stated time period and financial constraints. Closely-knit teams of experienced programmers and team leaders could reliably produce smaller programs which worked [Brooks 75, p.4], although their production process did not conform closely to the traditional software life cycle model, but larger projects consistently over-ran or failed to perform correctly. Perhaps the most well-known example of this was the attempts to develop the IBM 360 series operating system [Fishman 82]. The 16th release of OS/360, in July 1968, contained almost a million instructions, with the Fortran compiler alone containing 2000 separate errors [Randell 79].

In 1962 Bremermann [1962] applied quantum mechanical arguments to conjecture that no computer of any form (i.e. animate or inanimate) could process more than 2×10^{47} bits of information per gram of its mass per second[1]. The design and implementation of large systems, however, lies far from the outer limits of computational tractability. As it was clear that the problem lay not with the hardware designers, who were already making systems whose computational power exceeded the needs of programmers, much attention was focussed on to the programming side of systems. By the late 1960s, this study of the process of developing programs had been termed *software engineering*.

A number of reasons were advanced as the causes underlying the software crisis, including bad management, lack of facilities, lack of experienced programmers, etc. As an example, it was (quite naturally) thought that there would be some form of linear relationship between the number of people working on a system and the time it would take to produce the system. This, however, was found not to be the case [Brooks 75, p.16]. In fact, it was found that the restructuring of teams, the supply of extra resources (people, computers, administrative help), the hiring of

[1] which means, as pointed out in [Hayes 88], that a computer with the same mass as the earth (6×10^{27} grams) operating for a period equal to the estimated age of the earth (10^{10} years), could process less than 10^{93} bits of data. For comparison, a computer with 10^7 bits of addressable memory has $2^{10^7} \approx 10^{3,000,000}$ possible configurations, and there are approximately 10^{120} possible sequences of moves in chess.

expert consultant programmers, and other restorative measures failed to alleviate the problems and produce working systems. Similarly, the causes of the delays and cost over-runs could not be attributed to any one particular phase of the traditional software life cycle model.

The attention of some practitioners turned to more abstract methods of reasoning about programs. Each successive *generation* of computer technology has brought advances in understanding of how to properly utilise a stored program computer, and a number of approaches have been found fruitful. Attempts were made to develop programming methods which could aid the programmer in the efficient development of correctly working programs. These approaches can be split into techniques and paradigms.

The techniques of programming which were developed included high level languages, structured programming, modular languages and abstract data typing. The new tools that became available helped with compiler construction [Johnson 83] and application developers (by making use of fourth generation languages). For instance, progress has been made in the field of formally specifying what a program should do without recourse to the ambiguity of an English description, verifying a computer program with reference to a specification, automatically translating a specification into an implementation, etc. Such techniques are encompassed in the phrase *formal methods.*

The second approach to the development of correct programs involved the creation of different programming paradigms. The first high level languages, such as Algol, Fortran, COBOL, BASIC and C, were all based on the von Neumann model of computation. Pure LISP, a functional language, was released in 1957. Other paradigms include object-oriented and logic programming. In Chapter 2 we introduce a definitive programming paradigm. Selection of the most appropriate programming paradigm is an important component of software construction.

§1.3 Parallel programming is very difficult

In this section we outline the difficulties posed by parallel programming, and current approaches to dealing with these problems. The purpose of the section is to delimit the areas addressed within the body of the thesis.

The first form of concurrent activity in a computer was introduced as a result of the high cost of processing power in the 1950s and 1960s. Because of the processing time being wasted, there was a motivation to design special-purpose processors, including channel command processors (e.g. IBM 7000 series, 1959) and front-end processors, to autonomously handle interaction with the outside world, either peripherals or humans. *Time sharing* operating systems schedule work for the processor to perform when one process[2] is blocked, i.e. waiting for input. This technique of time sharing gives rise to a *multiaccess system*, and has the result of freeing the central processor for other, more computationally intensive tasks.

A characteristic of this early approach to concurrency is that each process had its own disjoint area of memory, and interaction between processes was mainly for the purposes of synchronisation. Therefore the fastest execution of a single process (typically a complete program) would be the time it would take the processor if that were the only process it was executing. It was soon realised that the only way to allow processes to run faster than the limitations imposed by a single processor is to allocate several processors for the execution of the program. Such a system is known as a *multiprocessing* system if all the processors reside locally enough that they can use memory as a shared *resource*, and a *distributed* system if information can only be exchanged over a data communications network of some form. The term *parallel processing* has been coined to describe any such system where more than one processor can be active amongst a group of processors at one time. Hardware advances have today made genuine parallel processing a reality in such machines as the Butterfly™ [Butterfly 85] and the Cosmic Cube [Seitz 85]

Problems specific to the development of parallel programs arise at both low and high levels of abstraction. The low-level difficulty arises in trying to formulate how two concurrently active processes will interact during the computation. For instance, in a language which permits changes in the value of variables, it must be ensured that no two write operations can be concurrently performed on the same variable, since a spurious value would then result. This problem with concurrency is known as *interference*, and interference avoidance typically involves identifying when the same variable is accessed by different portions of code. *Data dependencies* indicate when two operations must be done sequentially, because one affects a value consumed (i.e. used) by the other. We address the difficulty of identifying and respecting data dependencies in Chapter 2.

Other low-level concurrency-related problems include deadlock, deadly embrace, process control and scheduling, and by the end of the 1960s most of these problems had been identified. Techniques have been developed for describing the mechanism by which parallel activity is permitted, such as multithreading with a fork command (as used in the UNIX[3] operating system), critical region techniques (e.g. mutual exclusion semaphores

[2]It was in the early 1960s that the word *process* was first introduced. [Denning 71, p.171] described it as an abstraction of the activity of a processor, which allowed the concept of "program in execution" to have meaning at any time, regardless of whether a processor was actually executing instructions from a program at that point.

[3]UNIX is a registered trademark of Bell Laboratories

[Dijkstra 68]), and monitors [Hoare 74]. Other methods involve communication down dedicated lines, such as pipes and channels. As an example, the CSP specification language [Hoare 78] removes the burden of maintenance of the mechanism for parallel communication from the programmer by introducing input and output channels, thus focussing attention on the actual communications being made within the program. The abstractions and assumptions which are typically made in modern approaches to analysing the behaviour of concurrent programs are considered in Appendix 1. We do not directly address these other low-level problems of concurrency within this thesis.

High-level problems specific to parallel programming arise in considering how to write programs for parallel machines. Most programming of parallel machines is done using modified versions of sequential languages [Baldwin 87]. It is difficult using a modified sequential language for humans to describe in a natural way how a concurrent system is to behave. In Chapter 5 we describe how definitive programming can be used to give a more cognitively oriented method of programming, whilst still retaining a clear semantics which allows formal analysis.

The process of taking a sequential program and dividing it up into processes which can be run in parallel is called *parallelisation.* It is complex to analyse data dependencies in a sequential program, and therefore difficult to parallelise efficiently. There are two possible methods for the parallelisation of a program:

- augment the sequential language to allow the programmer either to supply information to indicate which parts of the program can be executed concurrently, or to explicitly write the code for each processor.

- mechanically identify the parallelisable elements of a program, and thus write compilers which will automatically transform a sequential program into one which can be efficiently run on a parallel machine.

The first approach puts the onus for finding suitable parallel processing on the programmer, which only serves to add to the difficulty of producing correct code. In practice it is found to be more difficult to write and debug parallel programs than sequential ones, due to the difficulty of maintaining a coherent and consistent idea of the state of execution, lack of design tools, etc. Inherent limitations of existing paradigms make the second approach of mechanical generation of parallelised programs difficult with current techniques.

We now give a taxonomy of current approaches to programming, giving a brief outline of each and commenting on their susceptibility to parallelisation. This allows an examination to be made of programming languages at an abstract level, rather than being concerned with specific details of implementation, syntax, etc. Existing programming languages can be broadly divided into two categories: procedural and nonprocedural. Procedural languages include von Neumann, object-oriented languages and constraint-based, and nonprocedural include functional, dataflow, and logic languages.

§1.4 Procedural programming

A procedural program explicitly describes the computation which is to be performed, but only implicitly defines the properties which the result is expected to exhibit. The description of computation is given in the form of a series of changes of program state, which must be executed in a predefined order, i.e. sequentially. This ordering of execution is important because the resultant state reached after a single computation (i.e. one program instruction) is a function not only of the instruction, but also of the state within which it was executed.

§1.4.1 von Neumann languages

Von Neumann languages are procedural languages which execute on a machine based on a von Neumann model of computation. A machine of this form contains a central processing unit, an arithmetic and logic unit, store, and input and output (I/O) facilities. Programs and data share the same store, and programs consist of a set of instructions to be executed sequentially. The program may use variables, labelled memory locations which can change their value in the course of program execution. Von Neumann languages include Pascal, COBOL, **C** and Fortran. Many useful concepts have been introduced for such languages, including structured programming, strong data typing, modularity and data hiding. Parallel von Neumann languages exist, notably occam, which allow the programmer to explicitly delimit code segments which can be executed concurrently.

Computation is state-based. The state of the system is given by the current value of all variables, and a transition between states is effected by executing an instruction. The goal of a terminating von Neumann program (i.e. one written in a von Neumann language) is expressed implicitly by specifying the transitions to be performed. In terms of a definition of a specification as being the "what is to be done" description and a program as being the "how it is to be done" description, the gap between a specification and a von Neumann program is considerable.

Procedure calls can have side effects. Side effects can take several forms: changes in the value of reference parameters, changes to non-local variables, performing input or output, and calls to storage allocators. Side effects make mathematical analysis of von Neumann programs complex, although an attempt has been made to categorise side effects into malignant and benign by Lamb [Lamb 88]. This corresponds to permitting several interfaces to a module, and specifying a side effect as being benign when it invisible to a client limited to that interface.

There may be components within the system which have only transient existence, such as allocated memory store in a procedural program, or instantiated objects in an object-oriented program. This differs from the nonprocedural approach in that all components of the system, such as functions and equations, are in use throughout execution. Definitive programming also uses transient components.

There are two forms of parallelism that can be abstracted from a procedural program: data and instruction parallelism. Data parallelism involves the processing of a number of items of data in parallel. An example of this is the code:

```
while(condition) {
    input(x) ;
    process(x) ;
    print(x) ;       }
```

This portion of code can be parallelised to allow several items of data to be input, processed and output in parallel, provided that the procedure calls do not have side effects. As the amount of data to be processed grows, so does the speedup obtained by data parallelism **-** there is no upper limit.

Instruction parallelism involves the execution of a number of instructions in the program concurrently. An example of this is the code:

```
a := x + (y*z) ;
b := x + w + (s * t * u) ;
c := w + x + y + z ;
```

which can be parallelised so that all three assignments are performed in parallel. As Hillis and Steele [1986, p.1182] observe, the use of instruction parallelism in a program of **n** instructions allows a speed-up by a factor of O(**n**) in the best case, i.e. when there are no data dependencies to impose any serialisation on the instructions. The allocation of more processors for a problem will, given enough data, always be useful for data parallelism, but with instruction parallelism there comes a point where the addition of further processors does not decrease the execution time.

The major difficulty which has been found in the parallelisation of procedural languages is that of data dependency analysis. Static analysis of data dependencies is complex because of the unstructured way in which the system state can be changed by side effects. The analysis of data dependency must therefore be on a system-wide scale, which is not practical for larger systems. What is required is some way of defining explicitly the side effects caused by each instruction, so that data dependency analysis can be done locally, without consideration of the effects on the system state of other instructions executed in unrelated parts of the program. This explicit structuring of data dependencies is supplied by definitive programming.

A further problem associated with data dependency analysis is the problem of *aliasing*. This is the difficulty of knowing when two instructions refer to the same address in memory. For example, if `array` is an array of integers, then the instructions

```
array[index1] := f(a,b) ;
array[index2] := g(c,d) ;
```

can only be executed in parallel if the two aliases, `index1` and `index2`, have a different value. When the language involves pointers this increases the difficulties, since now an in-depth analysis of the program would be required to resolve the reference. Some work has been successfully carried out on the problem of recognising aliases in array references [Bannerjee 79] [Nicolau 85], but it does not generalise to the more difficult problem of pointer aliasing. It is a very complex task to perform a complete data dependency analysis on a program which can refer to the same memory area by different names, and the general alias recognition problem is known to be undecidable. This is a limiting factor in the mechanical parallelisation of procedural programs.

Data parallelism is complex. A set of data which is to have the same operation(s) performed on each element is typically iterated over using some form of loop. Whether the loop is data parallelisable depends on whether there are dependencies between the elements of data generated by each iteration of the loop. Data dependencies must take into account the problem of aliasing, but it is in iterative loops that aliasing is found most profusely. It is very hard to analyse the data dependencies between loop iterations.

An understanding of the data dependencies which exist in a program is necessary for instruction parallelisation, but this seems to be very hard to accomplish within a procedural framework. The side effects associated with an instruction are a function of the state within which the instruction is executed, so there is a high degree of sequentiality inherent in the execution of a procedural program. The only time when two instructions can validly be executed concurrently is when the end result (i.e. state) is the same whatever the serialised order of their execution, which occurs precisely when there are no data dependencies existing between the instructions, a complex condition to analyse.

§1.4.2 Object-oriented languages

The object-oriented approach to programming uses information hiding by partitioning the variables (which define the state of the program) into distinct subsets, which are then *encapsulated*, along with the procedures (*methods*) which access them, into objects. One aim of encapsulation is to serialise what might otherwise be potentially interfering actions on the same variable. Interaction between objects occurs through message passing, which allows an object to invoke a method in another object which might, for instance, change or print the value of one of the recipient's variables. Object–oriented languages include C++ and Smalltalk. These are sequential languages in the sense that each object in the system executes internally in a sequential manner. The execution of objects is autonomous, so there can be parallel computation in the system.

The current state of execution of the system is defined by the values of all the variables and the messages which have been received, or are in transit. Messages which have been received affect the system state because they may alter how an object handles subsequent messages. Messages may be in transit because, unlike the shared variable method of communication, there may be a time lag between invoking a message and the result being visible in the object, with messages possibly interleaved in an order different to that in which they were sent.

Object-oriented programming offers the ability to limit what can make changes to variables by making them local and private, and only accessible through their associated methods (i.e. procedures) in the object. The intention is to impose a structure on changes to variables, to make analysis and detection of data dependency easier. It is found, however, that analysis of the data dependencies is non-trivial even in an object-oriented environment. To analyse data dependencies requires some notion of the state of the system, and so the concern is no longer only with what variables have been changed, but also with what methods have been invoked. Putting aside consideration of messages which have been sent but not received (they can be handled using the technique of time-stamping), the difficulty in analysing what messages have been received is a problem involving synchronisation issues, which are implementation dependent.

The problem is compounded by the 'cascade' effect, whereby object A invokes a method in object B which, as a result, invokes a method in object C, which affects how object C handles later invocations of its methods. Suppose after object A sends the message to object B, it sends one to object C, invoking a method whose behaviour will be changed by the arrival of the message from object B. It is then a matter of message timings as to which will arrive at object C first. Identification of the side effects of a message call (i.e. how it affects the handling of future messages) is non-trivial, because of the difficulty of following these effects across supposedly inviolable object boundaries, together with the associated problem of alias detection.

Furthermore, there are cases where effective information hiding is precluded by the semantics of the application, for example when the interdependencies between variables are global. Typically, this would occur where the dependencies are context-dependent, so that they change in different states. There is no provision for this within object-oriented programming - the dependencies are "hard-wired" into the structure of the object. An example of this difficulty in successfully implementing information hiding occurs in modelling a football match. Typically, one player will be assigned to mark another, which involves staying as close as possible to the marked player. If the game was to be modelled in an object-oriented manner, then the obvious approach to data encapsulation would involve modelling each player as an object. An object would invoke a method to find the position of the other object (the marked player), and update its own position accordingly. However, at various stages of the game either player will be marking the other, so the dependencies between the objects will change. Close synchronisation between the marker and the marked player cannot be guaranteed unless certain assumptions are made about the nature (in particular the speed) of message passing in the implementation. In order to ensure synchronisation, one would have to model the pair of players as a single object. However, during a football match one player might mark several of the opposing team members. Information hiding is not well suited to applications which contain global and dynamic data dependencies, an issue which is particularly effectively addressed by definitive programming.

§1.4.3 Constraint-based languages

Constraint-based systems are based on the automatic or semi-automatic maintenance of relationships between objects in a system. They can be part of a procedural program or be the relationships to be satisfied in a nonprocedural program. A common use of constraints is in graphics applications, where the objects on the screen are inter-related in some way which must be maintained. The constraints are two-way, in the sense that if A and B are constrained by some form of relation between them then a change to either A or B is permitted, and the other must be updated in such a way as to maintain the constraint. A typical constraint will consist of two elements: a nonprocedural description of the nature of the constraint, and a procedural outline of how to maintain the constraint.

In the context of parallel execution of a constraint-based language, both A and B (in the previous example) can be changed. This gives rise to the possibility of interference, when A and B are changed in an incompatible manner. It may not be possible to tell which procedure to invoke to re-establish the context, and indeed there may not be an appropriate procedural action. However, the data dependencies can be inferred from examination of the constraint description. We exploit this feature in the use of definitions, which are similar to uni-directional constraints.

§1.4.4 Parallelism in procedural programs

In summary, it appears that there are problems inherently associated with the automatic generation of parallelised code from procedural programming languages. Both instruction parallelism and data parallelism are limited by the complexity of data dependency analysis, which is compounded by the problem of aliasing. As Glaser, Hankin and Till [Glaser et al 84, p.ix] commented on procedural programming, "*...the notion of a global state that may change arbitrarily at each step of the computation has proved to be both intuitively and mathematically intractable*". Definitive programming is also state-based, but the representation of changes to the state has more structure than in a procedural program.

§1.5 Nonprocedural programming

More mathematically based approaches to programming allow a more abstract view of computation. A nonprocedural program states what properties the required solution is to exhibit, without stating how the solution is generated. This means that the gap between a specification and a nonprocedural program should not be as large as that between a specification and a von Neumann program (§1.4.1). As nonprocedural languages only specify what is to be produced, and not how, there is no description of control flow. There is therefore no concept of the current state of computation in a nonprocedural program.

Any procedural program can be written in a nonprocedural manner. This does not mean that nonprocedural programming supersedes procedural programming, since this would neglect certain features which a state-based computational model offers. For instance, the nonprocedural approach is not well suited to applications involving interaction with the operating environment, because the concept of input does not fit naturally into the paradigm. Input is often modelled as a stream of data, which is evaluated in a lazy manner. In a functional program this might be regarded as a function over time, which returns different values to represent the input value at different times.

Backus [1978] distinguishes four aspects of a computational model: foundations, history sensitivity, type of semantics, and the clarity and conceptual usefulness of programs of the model. History sensitivity is defined to be the capability of a program to store information that can affect the behaviour of a later program. Pure functional programming [Backus 78, p.623], and by extension nonprocedural programming, is not history sensitive, since it requires a notion of state to preserve in some way a history of the evaluation which has been performed to date. Nonprocedural programming is not well suited to such areas as interaction, concurrency and design (e.g. CAD/CAM).

§1.5.1 Functional languages

Functional, or applicative, languages, are the most well-known of the nonprocedural approaches, and employ the mathematically well-understood notion of a function. Functional programs are based on lambda calculus (i.e. recursion equations), and characterise a function from the initial state to a final state. Functions are defined by the use of recursion, other functions, and values, which are in turn characterised as functions of other functions. Such programs can be strongly or weakly typed, and can employ a lazy or eager evaluation strategy. What all functional programs have in common is that all references to a variable are *referentially transparent*, i.e. without side effects. This means that once a function has been defined, all uses of it elsewhere in the program can have the value of the function substituted in its place. In other words, there is a notion of equality which respects substitution, such that the denotation of an expression in a program is always the same. As an example, consider a function definition of the form

```
f x = ( ¬ x ) ( g x y z )
```

In a program containing this function definition, all occurrences of `f(n)` can be replaced by `g(n,y,z)`. This is because `g(n,y,z)`, being a (mathematical) function, will always return the same value. This is not true for procedural languages since `y` or `z` might change value during the course of execution of the program, or evaluation of the function `g()` might have side effects which alter the value of `f()`.

The changing environment of a reactive system (Appendix 2) can make application of the pure functional programming paradigm, based as it is on static functions, convoluted to program and difficult to implement. As an example, Miranda [Turner 85] [Thompson 86b] is a referentially transparent functional programming language which uses lazy evaluation in the manner of [Cartwright & Donahue 84]. Thompson [1986a] describes a method of handling interaction by treating input and output as a list, and introduce combinators for interactions. These include iterative combinators `repeat` and `while`, choice combinators `switch` and `alt`, and sequencing combinators such as `seq`. Their purpose is essentially to simulate their procedural counterparts. Another technique discussed is the addition of a dummy value to the input list, to allow for proper synchronisation between input and output. Whilst Turner maintains that such techniques are still susceptible to a formal mathematical analysis, it is clear that they are some distance removed from the intuition underlying functional programming.

A function in a functional program is defined recursively. As Baldwin [1987, p.9] observes, "*...recursive definitions introduce apparent data dependencies involving parameters to or results from the recursion. Often the dependencies are spurious, in that the order they impose on the computation is irrelevant to its correctness. Distinguishing spurious data dependencies from vital ones, however, requires deep knowledge of the algorithm...*". The use of recursion therefore make data dependency analysis complex.

§1.5.2 Dataflow languages

The difficulties which have been experienced with implementing parallel programs on von Neumann machines motivate an examination of languages suitable for execution on non-von Neumann architectures. Dataflow languages are based on a model of computation which uses dataflow graphs [Karp & Miller 66]. In a dataflow program an operation might be performed when the operands it requires become available. In a von Neumann machine sequential control flow dictates what instruction is executed next. A dataflow program, by contrast, involves the flow of data around the machine, which leads to a high level of parallelism. Procedural concepts such as control flow and memory locations are avoided. Lucid [Wadge & Ashcroft 85] is a dataflow language.

The data dependencies for each operation can be inferred, so problems in the implementing of instruction parallelism are avoided. To exploit this parallelism, new forms of architecture have been proposed, such as a transputer-based machine called ALICE (Applicative Language Idealised Computing Engine) [Darlington & Reeve 81]. Packets of data, status information and the instructions to be executed on the data are added to a ring to which are also connected switching matrices. When a processor (i.e. transputer) becomes available, the data packet is allocated to it, and the instructions in the packet are executed.

It can be difficult to ensure the switching is performed optimally, so that locality of reference (which reduces the distance a result packet must travel) is ensured. An efficient switching mechanism must take into account the time it will take to transmit a result packet, which may require annotations to the program if an optimal initial allocation of data packets to processors is to be performed at compile time. Full use of the parallelism in a dataflow language can only be made by execution on a special purpose architecture. In §3.4 we also suggest a new machine architecture to permit full exploitation of the parallelism in a definitive program

§1.5.3 Logic languages

Logic programming languages are particularly suited to inferring knowledge, in particular relations, implicit in a knowledge base by means of search techniques based on goals which are to be satisfied. In their pure form they are nonprocedural languages, and a program consists of testing whether a specified clause is a theorem in the logic. By far the most commonly used logic programming language is Prolog [Clocksin & Mellish 84]. A Prolog program is formulated in terms of a set of clauses which represent the available knowledge. Goals are then supplied which need to be satisfied.

The clause order and goal order of a Prolog program will markedly affect the efficiency of execution, and may even lead to non-termination. These orderings dictate the order in which solutions will be found, and determine the search tree, by specifying the flow of control. The search tree indicates the different unifications which can be tried in an attempt to unify every term (i.e. every subgoal).

Pure Prolog is neither sufficiently efficient nor expressive for practical usage. A number of procedural notions have had to be introduced which make programs harder to analyse formally. These measures include the cut predicate, and the read and write predicates. The cut predicate is used to prevent searches proceeding further down a search path than necessary once it is clear that no solution will be found, or to limit the number of solutions found. It does this by dynamically pruning the search tree. The read and write predicates (known as extra-logical predicates) are used to provide an input and output mechanism, a prerequisite for practical programming. They work by side-effect, a procedural concept which makes it hard to reason about the program in a nonprocedural manner.

Logic languages are very naturally suited to describing data parallelism. They express relations, not functions, so "*it is no harder to describe data parallelism over "outputs" than over "inputs", simply because these terms have no real meaning to the language itself.*" [Baldwin 87, p.14]. Logic programs are written at a relatively high level of abstraction, which often avoids the introduction of spurious data dependencies, as found in loops or recursion.

As with procedural programs, two major forms of parallelism have emerged: explicit and implicit. Explicit parallelism is derived by augmenting the basic logic language with concurrent constructs. In general there is a concomitant obscuring of semantics associated with these languages. Examples include GHC, Concurrent Prolog and Parlog.

Implicit parallelism takes advantage of parallelism which is implicit in the way searches are performed by, in theory at least, searching each part of the search tree in parallel. This has the advantages that there is no change in the syntax or semantics of the program and there is an improvement in efficiency. In other words, all the parallelisation is performed by the compiler, without any details being supplied (in the form of annotations or extra concurrent constructs) by the programmer. The possibility of shared variables in different subgoals makes parallelisation more difficult. Implicit parallelism can be subdivided into AND and OR parallelism.

OR-parallelism involves matching a single goal to many clauses simultaneously. This approach has several inherent drawbacks:

- the possibility of a large number of clauses being matched with a single goal means that there may be a large processor overhead,

- if each process has to store the binding environment, then there may be a large storage requirement,

- because the granularity of the search has been reduced, many of the processors may be underused,
- all solutions are looked for concurrently, even if one would be sufficient.

AND-parallelism involves the simultaneous resolution of several goals in a clause. The major problem associated with it occurs when there are binding conflicts, which result from two processes independently binding a shared logical variable to different values. One solution to this is to use restricted form of AND-parallelism, which only permits one goal to be active out of each set of goals which share a variable. This gives rise to the concept of a dependency graph, which specifies what goals must be satisfied before others can be matched. Another solution is to allow unrestricted AND-parallelism with "intelligent" backtracking when a binding conflict is detected. An example of this approach is the use of culprit sets, which combines unrestricted parallelism with dependency-directed backtracking. Other techniques for parallelism, such as low-level parallelism and instruction pipe-lining, have also been considered.

§1.5.4 Parallelism in nonprocedural programming

A nonprocedural program theoretically consists of a set of equations, formulae or functions which are sufficient to characterise the nature of the required result, without describing the evaluation strategy to be employed. It therefore appears that nonprocedural programs are easier to parallelise because, without the procedural notion of assignment, there are no side effects or aliasing problems to be dealt with. All that need be done is to divide up the work that needs to be done, such as function evaluation or satisfaction of predicates, into component parts of the required granularity. In practice, however, such a description of nonprocedural programs is incorrect: for reasons of both implementation and efficiency, the pure nonprocedural program must be augmented with other concepts which reduce the ease of parallelisation.