Automated source-to-source parallelisation of Fortran77 legacy code on shared memory computers

Craig Eldershaw
Department of Computer Science,
The University of Queensland,
Queensland 4072, Australia.
craige@cs.uq.oz.au

Semester II, 1996

\footnote{Copyright \textcopyright{} 1996, Craig Eldershaw
  typeset using \LaTeX}
Abstract

With the approaching limit on the speed increase of single processors, programmers need to look to parallel architectures for any further speed increases. Unfortunately parallelising code is a laborious and error-prone process. In the past there has been some work done on devising programs to aid programmers in the parallelisation of their code. However most of these require some degree of user interaction, and many lack portability.

The techniques outlined in this report are designed to yield portable parallel speed-ups in a completely automatic fashion. In particular this work focuses on the benefits possible through the “spreading” of Fortran77 DO-statements across multiple processors. To show the practicality of these methods, a pilot system was developed. Many of the issues involved in the automatic parallelisation of Fortran code are explored as part of the discussion of the design and effectiveness of this program.

The parallelised output of a number of different input programs has been tested on a shared memory machine. The results of some of these are given in this report.
Contents

1 Introduction ................................................. 4
   1.1 Existing programs ....................................... 5
   1.2 Limitations of existing programs ....................... 9
   1.3 Decisions and restrictions made in developing the pilot system 10

2 Parallelisation techniques/strategies .................. 12
   2.1 General strategies ..................................... 12
   2.2 Parallelisms in Fortran77 .............................. 13

3 Pilot study—an overview ................................. 15
   3.1 Study objective ......................................... 15
   3.2 Structure of the pilot system ............................ 15

4 The Fortran77 parser ........................................ 18
   4.1 Introduction ............................................. 18
   4.2 The Role of lexical analysers ............................ 18
   4.3 The Compiler-compiler: Bison ........................... 19
7 Examples and timing results

7.1 Analysis demonstration ........................................ 47
7.2 Speed tests ..................................................... 54

8 Conclusions and future directions

8.1 Overview of achievements ....................................... 57
8.2 Future directions ................................................ 58
8.3 Benefits and applications ....................................... 58

Bibliography

Acknowledgments
Chapter 1

Introduction

With the limit on the speed increase of single processors becoming closer, programmers need to look to parallel architectures for any further speed increases. Unfortunately parallelising code is a laborious and error-prone process. In the past there has been some work done on devising programs to aid programmers in the parallelisation of their code. However most of these require some degree of user interaction, and many lack portability.

Large amounts of the early Fortran77 legacy code are still in use. However in many cases, the original designers have moved on, and the current users do not know how the program works. Due to the size of these programs, reverse engineering and then re-coding of them is a very large and expensive task. It is therefore very desirable to find some way of speeding up these very old, but still useful programs in a way that does not require an understanding of how they work. One critical feature of any parallelisation process is for it to require minimal user interaction; to be as near as practicable completely automatic in its analysis and code generation. A paralleliser which detects plausible parallelism, and then queries the user is of far less value, since, while it saves some effort, the user is still compelled to learn/understand at least some parts of the program. Those parallelisers which rely upon special embedded compiler directives (eg. C$DO ACROSS) are similarly limited.

A further concern is portability. Fortran77 is a well accepted standard. Any Fortran77 code that runs on one machine, will usually run on another
with little or no alteration. This very useful property should be preserved by any parallelising process. That is, the resultant program should also be portable to any machine. In practice, certain practical limits need to be imposed, the most obvious one being based on general machine architecture. The work in this report focuses upon parallelisation techniques for thread-based shared-memory machines. So in the case of the pilot system provided here, the resultant code should compile and run correctly upon any shared-memory machine, but not necessarily upon any distributed-memory machine.

1.1 Existing programs

A number of systems have been developed within the last decade which attempt to address the problem of automatic parallelisation. Some are described (briefly) in this section, with their limitations analysed in section 1.2.

1.1.1 SUIF

A group of researchers at Stanford University are part way through a major project in optimising compiler design. There is a front end which converts the input code into a custom-designed, language-independent format. This format is given the name of SUIF (Stanford University Intermediate Compiler Format) and is also the name of the entire project[15]. The SUIF toolkit is comprised of a variety of compiler passes which can be run in sequence—each step hopefully improving the efficiency of the code. After all the passes have been run, a “back-end” to the suite converts the now optimised SUIF code back into C code ready for compilation by a native compiler.

One of the compilers can automatically generate parallelised code. It translates sequential programs into single-program, multiple-data (SPMD) programs that contain calls to the SUIF run-time library and runs upon shared address space machines. The parallelised code should run on any platform to which their parallel run-time library has been ported, however development testing has centred around SGI and Stanford DASH multipro-
cessor machines. Their library contains support for a number of different communications methods, but of particular interest is the use of POSIX threads (Pthreads).

The system provides many features to support parallelisation: data dependence analysis, reduction recognition, a set of symbolic analyses to improve the detection of parallelism, and unimodular transformations to increase parallelism and locality. The initial implementation of the system did not have a Fortran-to-SUIF and SUIF-to-Fortran conversion tool. The best that could be done was to use an enhanced version of f2c to convert Fortran77 into augmented C, and then to use the SUIF-to-C tool in the final step to generate C output. A recent enhancement (subsequent to the start of this University of Queensland research project) has provided a SUIF-to-Fortran-and-C tool.

1.1.2 Adaptor

The German National Research Centre for Information Technology has created a tool called Adaptor (Automatic DAData Parallelism TranslatOR) for elegantly converting HPF (High Performance Fortran) and Fortran90 to parallel Fortran77[20]. Any parallel directives in either HPF or Connection Machine Fortran style will be interpreted correctly. The resulting Fortran code does not involve explicit message passing at the high level, but rather uses a set of library routines to perform accesses to (possibly remote) portions of data structures, and to synchronise all copies of variables. A number of different versions of the libraries have been implemented, thus allowing the same high level code to function correctly in a number of different situations. These include: PVM, MPI, shared memory (using RPCs—Remote Procedure Calls) and a number of proprietary platforms.

1.1.3 Sage++

A group at the University of Indiana has created a tool known as Sage++ [14]. Sage++ is an object-oriented toolkit for building program transformation systems for the Fortran77, Fortran90, C and C++ languages. It is intended for use by researchers interested in building parallelising compilers, performance analysis tools, and source code optimisers. Sage++ is designed
as an open C++ class library that provides the user with a set of parsers, a structured parse tree, a symbol and type table and access to programmer annotations embedded in the source text. Central to the system, is a set of routines which allow the programmer to transform the structured parse tree before calling an “unparser”. So this program does not actually do any parallelisation—it merely provides a useful tool for the writers of a paralleliser.

1.1.4 Parafrase-2

The Centre for Supercomputing Research and Development at the University of Illinois at Urbana, has a compiler known as Parafrase-2[18]. Parafrase-2 is a vectorising/parallelising compiler, implemented as a source to source code restructurer. It consists of Fortran/C front-ends along with passes for analysis, transformation and code generation. However it appears to only assist with analysis, transformation and “unparsing”—it does not actually perform automatic parallelisation.

1.1.5 Partita

Cray Research Laboratories has an application known as Partita. Partita is a tool aimed at the analysis of parallelism within existing Fortran code. The tool analyses data dependencies in Fortran77 source code in order to achieve better loop parallelism. Partita is actually a module within their larger reverse-engineering system known as FORESYS. Despite being manufactured by Cray, the parallelised code resulting from the analysis performed is in fact platform independent (as opposed to using some kind of Cray-specific output). This platform independent parallelisation is due to the output being in Fortran90 code—not Fortran77.

1.1.6 Forge-90

Applied Parallel Research Incorporated has written a commercial program called Forge[16]. It generates a SPMD-style program using either MPI (Message Passing Interface) or PVM (Parallel Virtual Machine) based calls. This
paralleliser is primarily intended for use on IBM SP2 machines. However, due to the portable nature of PVM and MPI, its output should run on any distributed parallel system. The program is an interactive one which suggests a number of alternatives at each step and requests that the user choose the most appropriate one.

1.1.7 ParaScope

ParaScope is the result of a research project at Rice University (an important centre for this field of research)[17]. It does not actually parallelise the input Fortran code, but rather, provides interactive assistance to the user in restructuring serial code. The resultant transformed code is designed to be more amenable to parallelisation.

1.1.8 KAP

Kuck and Associates have created KAP, a commercial source-to-source optimising/parallelising pre-processor[13]. The KAP scanner recognises Fortran or C source code including a number of vendor specific dialects. It analyses the input code and automatically re-writes the code ready for parallelisation (for example, changing loop ordering in the case of nested loops if this will allow better parallelisation). The parallelisation component of KAP just restructures the code appropriately and then inserts compiler directives. Several versions of KAP are re-sold by some of the big-name computer companies. For example, the Power Fortran Analyser and the Power C Analyser distributed by Silicon Graphics are based upon KAP.

1.1.9 The Fortran Parallel Transformer

Two researchers at the University of Ghent in Belgium have developed a program called the Fortran Parallel Transformer[11]. It has been designed to: detect possibly parallel loops automatically; interactively perform transformations on the loops found; and then generate appropriately parallelised code. The code generated can run on several architectures by using message passing protocols such as PVM, MPI and some proprietary methods.
Alternatively, the code generator can be directed to produce code to run on shared-memory based multi-processor systems. The resultant program works by the initial process forking to create a specified number of copies of itself. Communication is performed using shared memory accesses and semaphores which are both supported by System V Unix.

1.2 Limitations of existing programs

Of the systems listed above, the ones which best match the stated objectives are the SUIF and KAP systems. Both of these work in a fully automatic manner (ie. they are genuine compilers, as opposed to interactive parallelising assistants), freeing the user from understanding the details of the code. The interactive component which exists in most of the other systems is a significant drawback.

The KAP systems, whilst effective at deducing parallelism in code (and making significant code-transformations where necessary), unfortunately are not designed to produce portable code. Instead they simply insert directives into the code and leave the actual parallelisation to the proprietary compiler. The directives are only usable by that compiler.

The SUIF compiler system on the other hand, generates code which makes calls to its portable library. One of the implementations of this library makes use of POSIX threads (sometimes known as Pthreads). This is a lightweight threads library which is highly portable. There are versions which will compile and run for almost all shared memory machines. However it does not produce pure Fortran code, but rather a combination of C and Fortran. This characteristic is unfortunate, because on many platforms, the Fortran compilers produce far more efficient assembly/object code than the C compiler will produce from the equivalent C code. This characteristic is partly due to the longer existence of Fortran, and consequently the Fortran compiler technology has had time to progress further than its C counterparts. In addition, the far simpler nature of Fortran data structures and the means of accessing those structures, gives the Fortran compiler far greater flexibility in optimisation since more limits are placed upon the tasks that the input code might be trying to do. When this pilot project started, no SUIF-to-Fortran converter existed; the addition of a SUIF-to-Fortran-and-
C converter is a very recent development—this time factor has prevented determination of the ratio between the C and Fortran codes components produced, and analysis of the efficiency of the interface between them. However, for reasons that will be described in section 6.2, a certain amount of non-Fortran code is necessary, whether this be in the form of C code (as for SUIF) or in non-Fortran libraries (as used in the KAP systems).

Various performance tests have been performed to compare the parallelisation abilities of KAP and SUIF. These indicate, in general, that the two systems produce comparable results, although each program performed better in some trials and worse in others due to their different “methods of attack”.

1.3 Decisions and restrictions made in developing the pilot system

The bulk of this report discusses a new implementation—a pilot system intended to demonstrate techniques and problems associated with parallelisation. The system acts as a source-to-source pre-processor. Ideally, any correct, serial, Fortran77 code is accepted as input, and correct parallelised Fortran77 code is output. In this output, all parallelisms which can be obtained, and would result in a real-time speed-up, should have been performed.

However, due to time and space considerations, the pilot system will not support all possible input Fortran programs. This constraint serves to simplify the code parser somewhat, but more importantly, it restricts the input, excluding some classes of programs which would require excessive work in the analysis stage. Even given that not all Fortran programs are considered valid input, this program will still not guarantee to be able to parallelise all loops which can be validly parallelised—this limitation occurs in all commercial implementations as well. Finally, to aid in keeping the analysis section to a reasonable size, the pilot system will only guarantee to correctly parallelise a subset of programs. That is, the code resulting from some input will not, in fact, be correct. However the subset of programs that can be correctly parallelised is well defined and is sufficiently large, to not only demonstrate relevant points of automatic parallelisation, but also
to be of some practical use. Details of the limitations of the pilot system will be addressed in precise detail in later sections.

Parallelism is achieved through the use of POSIX threads[19]. This has several benefits, but most importantly, it is an accepted standard listed by the International Standards Organisation. This enables the code which uses such calls to be transparently ported to any machine where the manufacturer has implemented Pthreads. These manufacturer implementations are usually highly efficient (since they can exploit low-level hardware/operating system specific actions). There is also an un-optimised free version which is portable to many platforms. In this way, the output of the paralleliser should be highly portable in a manner similar to programs using the BLAS and LAPack libraries. Highly optimised versions of these libraries exist on many systems, which allow the program to exploit cunning “tricks” which might exist uniquely on that hardware/operating combination. However in the absence of a platform specific implementation, generic, un-optimised versions also exist.

Pthreads are also preferable to several other, similarly portable, options. By their nature, Pthread-based programs are very scalable at little cost. This contrasts with the less efficient spawning of multiple program copies used in PVM, MPI or forked processors communicating with Berkeley-style Inter-Procedural Calls (IPCs). This is particularly the case when, as occurs in this system, the concurrently running code portions do not require any interprocess communication during execution. They (the slaves) simply communicate with the master at the beginning and end of execution.
Chapter 2

Parallelisation techniques/strategies

2.1 General strategies

There are several different strategies for parallelising a given program. One, rather naive method, is to determine if two or more consecutive statements are independent. When independent, they can potentially be run simultaneously on several processors. While this would be relatively easy to test and implement for some statements, any statement involving sufficient computation to be worth parallelising would probably involve a subroutine/function reference (“worth” in this case means that the theoretical reduction in processor time is not outweighed by the overheads of initialisation and communications). Unfortunately interprocedural analysis is extremely complex.

A variation upon the strategy would be to find consecutive independent blocks of code (ie. a group of several lines), and assign each block to a different processor. This could give sufficient work to each processor to make the parallelisation worthwhile. However, while having two consecutive independent statements is quite plausible, having a large number of consecutive independent statements, let alone a large number of consecutive independent blocks, is unlikely. Unfortunately, without this, the parallelism found is not really scalable (scalable means that doubling both the problem size
and the number of processors should result in little or no change in elapsed
time). Even assuming that sufficient numbers of adequately sized blocks of
independent code did exist, finding these blocks would be quite impractical.
The parallelising program would have no way of determining what combi-
nations of begins and ends should be used to delimit the chunks, except by
exhaustive trials.

An alternative strategy, which is common in practice, is parallelising or
“spreading” of loops. This can be performed when all the various itera-
tions of a loop are mutually independent. When all iterations of a loop are
mutually independent, each can be allocated to a particular processor, al-
lowing some to be executed simultaneously. From an abstract perspective,
this is really the same as the above method of simultaneously performing
independent statement blocks, having first “unrolled” the loop in question.
Unrolling occurs when a loop is replaced with repeated copies of the loop’s
code; there are as many copies as there are iterations, and each has had app-
propriate substitutions of the loop variable made. Not only does this solve
the problem of scalability (as loops frequently have large numbers of itera-
tions), but it also makes the choice of chunk beginnings and ends trivial—it
just tests the entire code contained within the loop as a block. This gives
rise to a very natural parallelism.

2.2 Parallelisms in Fortran77

In Fortran, the spreading of loops means the spreading of DO-statements.
DO-statements, according to the ANSI standard, take the form of:

\[
\text{DO} \quad \text{label variable} = \text{expression1}, \, \text{expression2} \, [\, \text{, expression3}]\]

What the parallelising program should do is allocate each loop iteration to
one of the available processors. A question which remains is: How should
all the loop iterations be allocated to the available processors?

For any given program, there may be several ways of parallelising it.
Ideally one would choose the method which results in the most effective use
of parallel computation (ie. gives the greatest speed-up). Critical factors
which effect its efficiency are: the overhead incurred due to the parallelism,
communication costs, start-up costs, etc. These can usually be reduced by using the coarsest level of parallelism possible (ie. allocating large amounts of work to a processor as a discrete unit, as opposed to a number of little tasks at frequent intervals).

Coarse-grained parallelism, however, frequently causes poor load balancing. This is the situation when one processor is allocated far less work than another, and so remains idle at a synchronisation point whilst waiting for the more heavily burdened processor to “catch-up”. In the particular case of DO-loops, it can occur because some iterations of the loop are significantly faster to complete. This is most likely to occur if some kind of branching control structure exists within the loop, for example: IF (i.gt.n/2) THEN ...ENDIF.

Good load balancing is usually achieved by allocating the work in as small portions as possible—and preferably in a dynamic fashion. The standard way to do this is to have a single “master” process (which has a large list of small jobs waiting to be performed) “farming out” the jobs to various “slaves”. As soon as a slave indicates that it is free to do more, then it is allocated a new portion of work. However this strategy obviously also maximises the number of times any parallel initialisation costs/etc. are incurred; so to achieve the minimum elapsed time, some kind of balance needs to be struck.
Chapter 3

Pilot study—an overview

3.1 Study objective

This report describes a parallelisation system which has been written by the author. The primary intent of the pilot study is to demonstrate the validity of automatic parallelisation, rather than to produce a fully fledged paralleliser. So, for the sake of simplicity and clarity, a number of limitations have been placed upon the parser and therefore on the total system.

The structure of this report is designed so that descriptions of how various problems were encountered and overcome during the development of this program, will explain some aspects of automatic parallelisation analysis. Apart from such illustrations, this report will also prove that such a system can be built, and will give useful results.

3.2 Structure of the pilot system

Automatic parallelising systems are usually written in standard ways. The most common method has a parser which reads the entire program into a structured parse tree. This tree can then be manipulated in its abstract form, before mapping the now modified tree back into source code. This method has the advantage that it allows reversal of loops (etc.) to be done
quite easily at the abstract level. These kinds of manipulations can generally lead to better parallel efficiency as the code is completely re-written in a manner which will achieve the best parallelisation. Tools such as Sage++ (see section 1.1.3) are very useful for this method.

The method used in this study is somewhat more straightforward, in that it concentrates upon finding parallelism without restructuring the code around it in order to improve parallelism. This allows a clearer demonstration of how automatic parallelism can be achieved. More complex restructuring is discussed in several works, the most notable being Banerjee[2].

Figure 3.1: A diagram which shows the structure of the pilot system with all the flows of data between all the modules

There are three distinct phases in the pilot system (see Figure 3.1). Initially, parsing of the input code is done (largely) on a line-by-line basis. Information extracted about each line is stored in a structure for later use. The second stage is the analysis of the code; this combines the line-by-line information generated by the parser, into information about entire loops
(such as whether or not they are parallelisable). Finally, the code generation section takes this information and re-reads the input file, re-writing its modified contents to several output files as it goes. The output files are ready to be compiled into a single, parallelised, executable program.

More detailed descriptions of each of these steps are contained within the next few chapters.
Chapter 4

The Fortran77 parser

4.1 Introduction

The parser is the first phase of the automatic paralleliser; its responsibility is to read the input file, line-by-line, and record relevant information about each. The input file itself, of course, consists of Fortran77 code. By far the easiest way of creating a parser is to use one of a class of what is known as “Compiler-compilers”. These are programs which create a program which acts as a parser. The input for this compiler-compiler is usually some fairly high level language which acts, in effect, as a description of the language that the resultant parser is expected to parse. Output from the parser is a complex data structure containing information about each line of the input file.

4.2 The Role of lexical analysers

In practice, rather than just using a compiler-compiler to create a parser, programmers usually do the work in two steps. The first step is to create a lexical analyser; the second step is the parsing itself (using the output of the lexical analyser).

A lexical analyser is a program or code fragment which accepts a char-
acter stream as input and returns a stream of tokens as output. A token, in this context, means a symbol representing reserved words (eg. ‘double precision’), special characters (eg. ‘+’, ‘(‘), numbers (eg. ‘12.3e-4’) and identifiers (eg. ‘myvar’). In the case of identifiers and numbers, two things are actually returned: a symbol to indicate what it has found (ie. a number or an identifier) and the value.

Again, rather than designing a lexical analyser from scratch, it is common to use a computer-generated one. A very widely used, free, lexical analyser-creator is Lex. This has been improved upon in functionality (whilst remaining almost completely backward compatible) by the GNU (Gnu is Not Unix) group who call their program Flex (for Flexible LEXical analyser)[9]. Flex accepts an EBNF-like (Extended Backus-Naur Form) syntax description of all the various tokens that the resultant lexical analyser will be expected to recognise. It also has a means of assigning symbols and return values to each such pattern. When run, Flex generates a C function which can be called by a parser each time the parser wants a new token.

4.3 The Compiler-compiler: Bison

A widely used compiler-compiler is known as Yacc (Yet Another Compiler-Compiler). Yacc has also been extended by the GNU group to create a piece of software called Bison[4]. The C program made by Bison does not actually read individual characters, rather it calls the lexical analyser (usually created by Flex) whenever it needs the next token. So the input to be parsed (in this case a program) is treated as a series of tokens. The actual input to Bison is a description of the grammar in a fairly primitive form. Basically it allows a series of rules, each of which states that a certain non-terminal symbol is equivalent to one or more combinations of terminal and non-terminal symbols. Terminal symbols are those which are returned by the lexical analyser when called. Non-terminal symbols are other high-level symbols which are defined elsewhere in the file.

Using this technique, an entire program is represented by a single non-terminal symbol which can be recursively broken down into many base patterns by a series of rules. For example, a rule might be that a subroutine call in the input language has an identifier, followed by a comma-separated list
of expressions and variables. Each rule can also have an action associated with it. So when a particular rule is used in the parsing, then the action specified by the user will be performed by the parser. Now, internally, the parser is just a finite state machine with a series of states and look-up tables. For each state that it might potentially be in, the parser has a table listing which new state it should enter, depending upon the value of the next token read. Sometimes this table will also include an action (eg. a call to a user-defined function) to be performed when changing state.

4.4 Difficulties encountered

The first parser trialled for the pilot study was written using Flex and Bison. It was not until this task had been completed that testing could occur. Now the combination of programs generated by Bison and Flex usually results in a reasonably efficient, relatively conveniently created and complete parser. Unfortunately in this case, it did not, and testing quickly revealed a couple of problems.

The first of these arose because of the highly unusual nature of Fortran grammers. In Fortran, there are no reserved words; it is perfectly valid to have a variable called ‘if’. In addition, except for columns 1 to 7, spacing has no significance. Consider, to take a particularly relevant example, the three statements:

- DO_10_I=1,50
- DO10I=1,50
- DO_10_I=1.50

All three of these are perfectly valid statements to the compiler. The first two of these are DO-statements (loops) and are identical to a Fortran77 compiler since spaces are not significant. However the last one is different, it is an assignment statement which assigns the value 1.5 to the variable ‘DO10I’.

It is quite possible for a human to see the difference in function (although somewhat difficult—that [comma versus full stop] difference is a notorious
“bug-inducing” feature of Fortran77; computer folk-lore even blames it for the crash of one of the early satellites). But how can a lexical analyser see this? In both cases it will correctly recognise the ‘=’ as a token in its own right. But it cannot correctly interpret the preceding five characters as it cannot tell, after reading in the first three, whether it has a complete token or not (since the digit could be a continuation of an identifier, or it could be the start of a new symbol, a label). Because a lexical analyser has no notion of the surrounding symbols (it processes a continuous stream of data) then it cannot decide.

An alternative is to not use a lexical analyser, but instead make the parser work at a much lower level by providing it with a stream of tokens, each of which represents only one character. This leaves the parser (which does have knowledge of context) to determine which interpretation to use. This of course undesirably clutters the parser by forcing it to work at such a low level, but it does avoid the problem.

This is not a realistic solution; it simply moves the problem. While the parser has some contextual knowledge (such as historical knowledge, in that it knows what symbols have just been processed and what its current state is), most automatically generated parsers have only very limited look-ahead capability. Bison (and Yacc and indeed most compiler-compilers) can only support a subset of the languages that can potentially be described by the BNF input language used. BNF describes a context-free grammar, but Bison (etc.) only support LALR grammars. In practical terms, this means that at any given state of the parser, it must be able to determine what new state to go to by looking only at the next token.

If the situation occurs, that for a given state there is more than one way to leave, all of which involve accepting the same symbol at the next step, then the parser is confused. This particular problem requires several tokens “look-ahead” to determine what to do. Until the parser reaches either the comma or the end of the line, it cannot tell which way it should have branched several states ago. But by this time it is too late.

This particular example demonstrates the spacing-related problems (ie. if it was mandatory to have a space after a ‘DO’ keyword, and invalid to have a space in an identifier, then all would have been well). However other, more serious problems exist when parsing Fortran77. There are several different kinds of ‘IF’ statements and ‘GOTO’ statements (eg. unconditional GOTOs,
computed GOTOs and assigned GOTOs). Exactly which variety a particular line contains cannot be determined until part of the way through it—again bringing Bison to a halt. In practice, Bison arbitrarily selects one of the possible state changes, and then, when it encounters a token that is inappropriate for the decision made (e.g. it finally came across the comma in the DO-statement), it stops and reports an error in the input.

As a result of the above difficulties, the experiment with Flex and Bison had to be abandoned and an alternative found.

4.5 PreCCx

A solution to all of these problems was found (thankfully, although annoyingly late in the project). It came in the form of a significantly different compiler-compiler called PreCCx (Pretty Compiler-Compiler Extended)[12]. This program, developed in Oxford, is unusual in that it generates “infinite look-ahead” parsers. That is, having reached a state where two or more state changes could be made when one look-ahead is insufficient, it chooses the one indicated by the initial language description (PreCCx-generated parsers always start by trying the first branch listed). If it reaches an inappropriate token (if it had tried the assignment-statement interpretation, then the inappropriate token would be the comma), then it performs a “backtrack”, to where it last made an ambiguous branch, and then tries the next. When back-tracking, it stores the tokens which it had read after passing the branch point, and then tries to re-interpret them starting from the next possible transition-destination state.

Only after all possible avenues have been explored, and every avenue having been blocked by an inappropriate token, does it fail with an error. Consequently, as long as care is taken to place the rule which “expands the most” first, then this solves all the problems. That is, a rule that expands to ‘A B D C’ should go before one that expands to ‘A B D’ (where ‘A’, ‘B’, ‘C’ and ‘D’ are all symbols). This way, upon starting down the first path, if it reaches a non-‘C’ in the 4th place, then it can revert to the second. If the reverse ordering was used, then upon reaching the ‘D’, the rule would report that it was correctly completed, preventing back-tracking and then not being able to match any rule beginning with a ‘C’.
It is worthy of note that the PreCCx input language is far superior to that of Bison/Yacc. It allows virtually full EBNF declarations (including: optional; zero or more repeats; and one or more repeats). The actions associated with and the values returned by each action are also more flexible. However, it has some inconveniences, in that the parser it generates is intended to be run as a stand-alone package—not as a subroutine within a larger program as is desired here. Consequently, some non-standard use of the code generated is required for it to be made callable. Finally, while it is intended to easily interface with a lexical analyser in a similar manner to Bison, this was not found in practice. In fact it was so awkward, that a lexical analyser was dispensed with completely.

4.6 Further complications

There are still a few complications in the parsing. They are solvable using PreCCx, but are awkward. While most spaces can be ignored, some are important—those in the first seven columns. This means that the lexical analyser (which was replaced with a low-level fragment of the parser in the pilot system) cannot simply be directed to drop all white space, but rather is forced to pass it on as tokens. So now the parser has to be able to ignore all white space in the middle of key-words, identifiers and numbers. For example the word ‘precision’ should be defined using a rule something like ‘P ws* R ws* E ws* C ws*…’ where ‘ws’ is another rule that expands to any kind of white space (spaces or tabs).

Another difficulty that is solvable using PreCCx (although messy), is the support of comments and continuation lines. In many languages, one can determine the end of a statement by some marker (eg. a ‘;’ in either C or Pascal); however Fortran77 uses ends of lines (after a fashion). It is possible, in Fortran to have continuation lines, that is, lines whose first five columns are empty, but the sixth is non-empty, which implies that the rest of the line is actually a continuation of the previous line. While this is somewhat awkward, what really causes difficulties in parsing, is the fact that one or more comment lines can be intermingled between the start of a line and any of its continuation lines, so that after every character in every part of an expression, there is the possibility that several comments may exist, compounding the problem. Remember that in parsing these
comments and continuation lines, some spaces—down to the exact column location—are relevant. As an example, the following is a perfectly correct Fortran program which declares \texttt{myvar} to be an array of 100 integers:

\begin{verbatim}
program silly
  in the C isn't parsing Fortran nasty
  &ger myvar(100
   &***************
   &0)

end
\end{verbatim}

4.7 Extracted information

As stated earlier, the information extracted during the parsing stage has been done on a line-by-line basis. This vastly simplifies the parsing stage by moving the collation of information about larger structures to the analysis stage. This is, it could be argued, the most appropriate place anyway. No limitation upon the working of the program is created by this. As each line is read, a record of information is added to a linked list. The record contains: the line number; the statement type (a \texttt{GOTO}-statement, a \texttt{DO}-statement, a \texttt{CALL}, or other); any labels referenced; the label of the line (if it has one); any variables referenced; the variable changed (if any) and whether that changed element is an array element or scalar; and finally, the start, end and step-size (if any) of any \texttt{DO}-statements. Note that \texttt{RETURNs}, \texttt{STOPs} and \texttt{PAUSE}s are also flagged as call statements.

In addition, if any variables are declared then their name and type is stored. Arrays declared are stored separately. When parsing an expression, a function reference is frequently indistinguishable from an array reference (eg. it is not possible to tell by inspection whether the code ‘\texttt{A=XYZ(C,D)}’ is accessing the two dimensional array \texttt{XYZ}, or calling the function \texttt{XYZ}). Differentiating between functions and arrays is one of only three places where any cross-referencing to previous lines is performed by the parser. When one of these ambiguous references is made, the parser checks back to arrays already declared. If it is not found in that list, then it is interpreted as a
function call, and in this case the line is set to be of type call-statement. This interpretation correctly conforms to the standard.

Whenever a constant is defined (using a `PARAMETER` statement), the expression assigned to the variable in question is evaluated and stored. If the expression is just a value, then this is trivial. However if the expression contains references to other constants, then a look-up is performed on previously defined ones. In any correct program, no expression assigned to a constant can make reference to: a variable; a function; or a constant not previously defined.

Finally, when a loop is encountered, an attempt is made to evaluate the upper and lower bounds and the step size (if present). This is relatively straightforward for straight numerical expressions. However if any constants or variables are involved then the process is more difficult. In the case of constants, the previously mentioned list is accessed and the stored values are retrieved. If a variable is encountered (any identifier not listed in the constants), then a flag in the line’s record is set to indicate that the bounds and/or step size could not be calculated at the compile stage.

4.8 Limitations and future directions

The primary intent of this pilot system is to demonstrate the validity of automatic parallelisation rather than to produce a fully fledged paralleliser. So, for the sake of simplicity and clarity, a number of limitations have been placed upon the parser and therefore on the total system.

The parser developed has almost no error checking because it assumes that the input code is correct Fortran code as defined by the Subset language specification. The ANSI standard for Fortran77 actually defines two languages[1]; one is termed the Full Fortran Language, the other is the Subset Fortran Language.

In addition, the parser does not support: character strings; continuation lines; `equivalence` or `implicit` statements; more than one block of code at a time; arrays with more than one subscript; spaces within keywords, identifiers or numbers; or input/output commands (`print`, `read`, etc.). Nor does the evaluation of expressions correctly support operator precedence.
Most of the changes required to remove these limitations effect only the parser. They could be removed by some simple extensions to the parser, although this would be at significant cost to code clarity. The relevant changes would involve:

- **character strings.** Apart from support within the parser, the addition of character strings can be achieved by the addition of another variable type in the interface to the parser/analyser/generator.

- **continuation lines.** Allowing for continuation lines will cause the greatest difficulty in extending the parser, and would require the LineRec record structure to be modified to support storing the first and last line numbers of any given statement. Related changes would also be required in the code generation section.

- **equivalence or implicit statements.** These statements are accepted and parsed, but no information about them is stored for later use. All variables and constants must be explicitly declared in this pilot system. Adding support for either of these would require alterations to be made to the parser (to store the information) and the analyser (to interpret it). In the case of the implicit statement, the generator would need to copy such lines, where relevant, into the output code.

- **more than one block of code at a time.** The pilot system’s parser stops after reading in one function, subroutine or main program block. To remove this limitation, changes could be made to the system to iterate the parse/analysis/generate phases for each code block.

- **arrays of higher order.** While the parser will correctly parse arrays of order greater than one, it does not save the details for later use by the code generator. This would require a modification to the data structure currently used to store information about arrays.

- **input/output commands.** The input/output commands do require a little more work as they cannot be run within a parallel section of code (because the ordering of such statements is critical, and such ordering cannot be guaranteed in a parallel section). However flagging this line as being unparallelisable would suffice—the analyser section would then decline to parallelise any loop containing input/output.
• expression evaluation. The evaluation of expressions (used to evaluate constant declarations and parameters to loops) is not entirely correct. In this pilot system, only integer arithmetic is supported. Furthermore, correct operator precedence is not implemented—all expressions are evaluated strictly left-to-right. To correct the precedence, that section of the compiler grammar would require restructuring. To allow full floating point constant evaluations, some flow-on changes may be required to variable types throughout the system.

4.9 Summary

The parser, phase one of the system, reads original, serial Fortran source code and generates a collection of records containing information about each line of code. This information becomes the input to phase two of the system, the code analyser.
Chapter 5

The Code analyser

5.1 Introduction

Parsing having been completed, it is the task of the code analyser to determine what, if any, portions can be parallelised. As previously stated, the pilot system attempts to achieve parallelism by spreading DO-loops. At the top level of the analyser is a loop, which looks at the record stored for each line in turn. Whenever the start of a loop is detected, a careful search is made of all the lines through to its end to determine if it can satisfy the necessary criteria for parallelisation.

The analyser takes as its input, the information contained within data structures prepared by the parser. It processes this and deduces information about DO-loops which can be parallelised. This information becomes the input for the third phase, the code generator.

5.2 Determining if a loop is parallelisable

There are a number of factors which govern whether a particular loop is parallelisable or not. In general, it can be assumed that a loop is parallelisable, unless it violates one of the conditions outlined below (sections 5.2.1 to 5.2.3).
5.2.1 Structural considerations

One inviolable condition for code to be parallelisable is that any GOTO statements must only have GOTO labels which are contained within the current loop. While the Fortran77 standard does not allow jumping into loops from the outside (as the current loop value would be undefined), it does allow program execution to jump out of a loop. This cannot be permitted in a parallelised loop situation. The problem lies in the sequencing of the actions. Take, for instance, a 20-iteration loop. If the 10th iteration of this loop contains a jump out of the loop, then iterations 11–20 should never be performed. However in the case of a parallelised loop, some or all of these iterations are performed simultaneously. So iterations 11–20 would have already started (indeed, some of them may have finished) before it is realised that they should not ever have been executed. To prevent this problem, for each GOTO statement in the loop, a check is performed to confirm that all the destination labels are within the scope of the loop. Since it is invalid to attempt to jump into a DO-statement with a GOTO, then this need not be checked.

Probably the most limiting of all the constraints on parallelisation, is the ban on subroutine/function calls. The basis for the ban is that the calling program does not know if the subroutine will change any of the values passed into it, or if it contains any input/output. This prohibition is normal practice; for example the KAP parallelising analyser refuses to parallelise any loop which contains a call unless it has a specific directive to “re-assure” it. However since the pilot system is intended to be entirely automatic, so as to avoid the user needing to know the workings/structure of it, then this “re-assurance” cannot be obtained. The SUIF team at Stanford do not currently support inter-procedural optimisation, but are working upon it. Since jumps out of the loop cannot be permitted (as explained above with regards to GOTOs), then the statements RETURN, PAUSE and STOP cannot be permitted either. This is easily prevented in this implementation, as any line containing one of these is flagged as a call-statement by the parser. So the analyser simply refuses to parallelise any loops containing a line flagged as a call statement.
5.2.2 Scalar variable considerations

Recognising various disallowed commands (see section 5.2.1) is by far the easiest to implement of the constraints to parallelisation. The analysis becomes more difficult when considering variables which are altered. The basic premise for parallelisation, is that all the iterations are independent. In particular, this means that information used in one iteration cannot be created in a previous one. So if the $i$th iteration alters or assigns a value to a variable, then (other things being equal) it is safe to parallelise the loop if and only if that variable is not referenced (ie. used in an expression in such a way that the value of that expression depends upon the current value of that variable) in the $j$th iteration, for any $i < j$. While this is a sufficient condition, it is far stronger than is necessary, making it a considerable impediment to parallelisation.

A more complicated, but far more effective condition is: if any variable is altered or assigned a value in the $i$th iteration, and is referenced in the $j$th iteration ($i < j$), then that variable must have a value assigned to it in the $j$th iteration at a time before it is referenced. While easily stated, this sometimes proves to be far more difficult to implement. For scalar variables, the procedure used in this pilot system is as follows. For each variable referenced on each line, two scans need to be done: one ranges from the start of the loop until just before the current line, to determine whether the variable has been initialised before this line; the other scan ranges from the current line to the end of the loop, also checking if the variable has been assigned a value. Then, if the variable is altered on or after the current line, but has not been assigned a value before this line, the loop is not parallelised.

Importantly, this technique is only guaranteed to be correct in the absence of control structures within the loop. Other loops (ie. nested loops) pose no problems as long as at least one iteration is performed—however IF-statements and GOTO-statements can result in false parallelisation. The above algorithm assumes that all lines of code are executed exactly once in every iteration. Furthermore, it assumes that they are executed in the sequence that they appear in the file. So if a line of code, which the above algorithm assumes will initialise a variable, is not executed every iteration, then this will not be detected, and the loop will be deemed to be parallelisable when it is not. Correcting this deficiency would be very difficult, as it would require a full, compile-time analysis of all possible program execution-
flows.

Of course it would be possible to simply reject all candidate loops which contain such IF/GOTO/DO-statements. However this would greatly limit the number of parallelisable loops. In particular, preventing the parallelisation of the outer of nested DO-loops would be extremely detrimental, as better parallelism in the case of nested loops is almost always achieved by spreading the outer loop. If the simplistic approach (rejection of loops that include DO-statements) were applied, then only the innermost of any nested loops would ever be parallelised.

5.2.3 Array/vector variable considerations

A situation characterised by arrays or vectors needs to be considered differently to one only involving scalar variables. Arrays (and vectors, which are simply one dimensional arrays) pose significant problems. This is because an array should not be treated as a single large variable, but as many small independent ones. That is, if one iteration varies one array element (eg. myarray(5)), and another iteration varies a different element of the same array (eg. myarray(2)), then these should be treated as separate variables.

One (rather limiting) solution would be to make the simplifying assumption that all array elements will only be referenced by integer constants (ie. no variables will ever be used in an index to an array). This would then allow the parallelisation program to treat each element as a single variable, and deal with it in the usual fashion (having first numerically evaluated each of the array indexes to a single integer value). However even in this case difficulties occur. For a significant sized problem (which is where a system such as described here would be of use), two (or even higher) dimensional arrays are probably used, so the number of independent variables “created” by treating each element as a separate variable in the normal way would rapidly become prohibitive. Take for example a 1000 by 1000 array. It would contain $1000 \times 1000 = 1,000,000$ variables which are to be processed, making the parallelising system’s task impractical.

However the most common use of DO-loops (indeed arguably the only reason for their existence) is to access a large number of array elements and carry out some operation upon each in turn. A simple example would be to
add two vectors together:

\[
\text{DO } 10 \text{ I}=1,100 \\
10 \quad A(I)=B(I)+C(I)
\]

In a case such as this, the central purpose of the loop is expressly to use a variable as a part of the index; which has major benefits. One benefit is that it saves the programmer from having to generate large, quite impractical programs; for example 1,000,000 lines of code could be required just to zero an array. In addition, the number of elements which need to be accessed at a particular point in time, or for that matter, even how many elements there are in the array, may not be known at compilation time, but are determined dynamically. An example of the first is \text{DO } 10 \text{ I}=1, \text{UsersGuess}. Here the program must iterate over a set of instructions \text{UserGuess} number of times, where that number will change each time the program is run with different input and so certainly cannot be known at compilation time. A case where the code cannot tell even what size the array is, occurs typically within a subroutine. For example in the subroutine:

```plaintext
Subroutine DAXPY(N,X,Y,A) 
Integer N 
Double Precision X(N),Y(N),A 
Integer J 

DO 10 J=1,N 
10 \quad Y(J)=A*X(J)+Y(J)

Return 
End
```

the compiler which compiles this subroutine does not know the value \text{N}—and hence the size of the array—until it is called by some other program component (which may not have even been written yet, since compiling can be done in a modular manner).

Variables must be allowed within array indices for any meaningful program to be written. However it is also obvious that to correctly determine
the parallelisability of a given loop which uses such indexes is an extremely
difficult task, since such determinations must be done symbolically. For ex-
ample, take the code in the middle of a pair of nested loops with variables
I and J. The parallelisation program must be able to determine whether
the two elements $X(I+(J-1)\times N)$ and $X(2\times M+J\times I)$ in fact refer to the same
portion of memory. If they do, then those two sections of code should be
treated as accessing the one variable, however if they do not, then the two
references to the array $X$ should be treated as accessing two completely in-
dependent variables. Such determinations become even more complicated if
the values of M and N are changed throughout the life of the loop (eg. if the
line $M=M+2\times I$ occurs within the loop also). A final complication can occur
in some situations, in that loops exist which are parallelisable for some val-
ues of variables, but not for others (eg. for $N>2$—where $N$ is some variable
whose value is set before entering the loop—the loop is parallelisable; but for
any other value of $N$, the loop should be deemed to have data dependency).

While much theoretical work has been done on this topic (known as data
dependence analysis), the complexity of the issue is such that potential so-
lutions cannot be discussed in this report. The interested reader is directed
to a number of articles published including [3, 5, 6, 7, 8]. For a very com-
prehensive treatment, the book by Banerjee is considered to be one of the
definitive works on the subject[2].

5.3 Benefits and limitations of the analyser

The pilot system uses the algorithm described in section 5.2.2. As stated
there, it is not guaranteed to work if certain structures are present within the
loop. There can be situations where a loop which otherwise meets all basic
criteria will be parallelised, but if such structures do exist, then incorrect
code may result. By incorrect code, it is not meant that the output will not
compile correctly—any correct input code which is accepted by the parser
will result in code which will compile—it is simply that the resultant code
may yield different results to that produced by the original serial code.

Due to the difficulties outlined in section 5.2.3, no data dependency
checking is done for array elements. All arrays are treated as shared objects
which may be accessed or updated within any or all loop iterations with
no locking or other protection. A feature which has been built into the pilot system for demonstration purposes, is a routine which attempts to evaluate loop bounds and step sizes to a single numeric value. It does this by substituting in the values of any constants used in the expression and then evaluating the resulting expression. If a variable exists within any of the two (or three, if there is a value for the step size) expressions, then the evaluation fails and a flag to indicate this is set. During the analysis section, any loop with non-constant bounds or step sizes is deemed non-parallelisable. This is not a useful heuristic in itself, but is included to give some idea of how constant expressions can be evaluated to single values; abilities such as this would be necessary in any comprehensive parallelising system which correctly dealt with arrays.

The net effect of the restrictions and limitations described above are:

- almost all loops with fixed bounds and step sizes which should be, are parallelised;
- no loop with variable bounds or step sizes will be parallelised, even if it can be done correctly;
- a number of loops which should not be parallelised as a result of array element interaction are incorrectly identified as being parallelisable; and
- a number of loops which should not be parallelised, due to data dependency with either arrays or scalars, but which contain flow control structures, are also incorrectly parallelised.

In spite of some limitations, discussed above, the pilot system is a success. It has demonstrated the feasibility of automated, portable parallelisation. Several examples are shown in Chapter 7.

### 5.4 Summary

The parser, phase one of the system, reads the original, serial Fortran77 source code and generates a collection of records containing information
about each line of code. The code analyser, phase two of the system, reads the line-by-line information from the parser and produces information regarding DO-loops that can be parallelised. This DO-loop information, along with the original Fortran source code, becomes the input to phase three of the system, the code generator.
Chapter 6

The Code generator

6.1 Introduction

The generator re-reads the original Fortran77 source file in conjunction with the output of the second phase, the code analyser. The latter is a data structure which contains information guiding the generation of routines to be run in parallel. The generator produces: a Fortran77 main program; Fortran77 subroutines which correspond to DO-loops in the original source code; and C routines which link the main program and the subroutines in a manner which allows parallel execution.

Of the three sections of the paralleliser, this is in many ways the simplest. The aim is to re-read the input Fortran77 file, while using the information deduced in the analysis stage, to create one or more output files. These output files should all compile and link without modification. When run, the resulting program should give identical output to that of the input serial program. For the reasons given in section 6.4, the output code cannot consist entirely of Fortran77 code. Most normal compilers do not support mixed languages within a single file, and so one is needed for each of the two languages. This pilot system chooses to use three files for “tidiness” sake.
6.2 Dynamic-allocation

There are several limitations in Fortran77 which do not occur in some other (generally more modern) languages. Two of the most fundamental of these are the absence of dynamic allocation of variables, and the invalidity of recursion.

Elaborating briefly upon the first point: Fortran77 makes no provision for creating variables of dynamic size at run-time. In languages such as C or Pascal, this is trivially done through the use of pointers and a few library routines (malloc, etc.). If the program needs an array of integers of size $N$ but does not know the value of $N$ until run-time, then in C one simply declares a pointer to a list of integers (int *my_var;). Then, at the point in the code where $N$ is finally known, a simple line such as:

```
my_var=(int*)malloc(N*sizeof(int));
```

will solve the problem. However Fortran77 has no concept of pointers, and so a program cannot dynamically allocate memory on demand—the dimensions of all variables must be known at compilation time at the location in which they are declared. That is, a subroutine which passes an array need not know its dimension, but it must be known at the location in which it is declared.

There are several problems with recursion. In the first instance, the Fortran77 standard specifically disallows either direct or mutual recursion (mutual recursion is where A calls B which calls A again). Moreover, as seen above, all Fortran77 code must contain enough information for the compiler to determine exactly how much memory will be used. While the standard does not dictate how memory allocation for variables should be performed, most compilers take advantage of this information to statically allocate all the memory that will ever be required by the application. While this may be inefficient from a memory consumption perspective (if two subroutines which will never call each other both require 10,000 bytes of local storage, then why should 20,000 bytes be allocated?), it means that no fragmentation of memory occurs and the overhead of growing and shrinking stacks, characteristic of most other languages, is not incurred. This means, that even if recursion were to take place, each copy of the subroutine which is called recursively would share local variable space. So if one of the recursive copies (the “child”) changes a local variable, its parent’s (the copy of the same subroutine which called the child) local variable is also changed.
To show how this works, a short recursive program and its output is shown below. Note that this code is not in fact valid Fortran77 because of the recursive call, but it is compiled by many Fortran77 compilers.

Program CrashIt

Call Recu(0)
End

Subroutine Recu(ind)
  Integer ind, inter

  inter=1
  Print*,"Ind",ind," inter",inter
  If (ind.lt.2) call recu(ind+1)
  Print*,"Ind",ind," inter",inter
  inter=2
  Return
End

The output is as follows:

Ind= 0 inter= 1
Ind= 1 inter= 1
Ind= 2 inter= 1
Ind= 2 inter= 1
Ind= 1 inter= 2
Ind= 0 inter= 2

Now if an equivalent program was written in C and run, all the inter values printed would be 1. This is because each local copy of inter is always printed out after it is set to 1, but before it is set to 2. However in the Fortran code, the child at depth 2 (ind=2) changes its parent’s shared inter value before returning.

Although recursion is not allowed by the Fortran77 standard and is not used in this implementation, a related issue does occur when running several
loop iterations simultaneously. This is basically achieved by extracting all
the code from within the loop and placing it within a subroutine which is
called several times simultaneously. Take for instance the loop:

\[
\text{DO 10 } I=1,N \\
\quad X= \text{dble}(I)/\text{dble}(N) \\
10 \quad A(I)=X*(X-1)/\text{sin}(X)
\]

This can be validly parallelised by placing the second and third lines in a
subroutine, but care must be taken. The problem is that in each of the
copies that are simultaneously running, the variable \(X\) should have different
values. This means that there must be as many copies of \(X\) in memory as
there are simultaneous iterations. A slightly weaker condition must occur
with \(I\): according to the standard, the loop variable cannot be altered by
any code within the loop, but each iteration does need to be able to obtain
the value of \(I\) in its iteration. It turns out, that this is most easily achieved
by maintaining a separate copy of \(I\) for each iteration as well.

In a language that supports stack based allocation of local variables,
one could simply have local variables \(I\) and \(X\) and the compiler would auto-
matically allocate as many copies as are needed. However many Fortran77
compilers insist upon using a single shared version. An alternative that
could be achieved in some languages, would be to dynamically allocate the
space needed for each iteration. However again, Fortran77 does not support
such a solution. So the only feasible process in Fortran77 is to pass all local
variables into the subroutine.

6.3 Pthreads

The Pthreads standard defines a number (twenty or so) of different routines
which can be called to start, stop, suspend and otherwise control the exec-
ution of various threads. Only two of these routines are used by the pilot
system: \texttt{pthread_create} which starts a new thread, and \texttt{pthread_attach}
which stops the parent program until the specified thread terminates.

Unfortunately, the Pthreads standard only allows one parameter to be
passed to the function which is started as a thread. As is explained in the
previous section, all local variables must be passed in, and in addition (unless common blocks are used), so too must all shared variables. So this one value passed to the called function/subroutine must effectively contain all the relevant variables. Fortran77 has no way of doing this as it has no concept of structures or pointers. While it could be argued that common blocks constitute structures, they do not suffice in this situation as the program cannot practically create a separate block for each loop iteration.

6.4 Mixing languages

Probably the simplest way around both of the problems described in sections 6.2 and 6.3, is to mix languages. A small component of another language which supports dynamic allocation, pointers and structures can overcome both difficulties. While there are several languages which would meet these criteria, C is probably the most appropriate (given that portability was one of the stated aims of this paralleliser’s output code). Furthermore, since most, if not all, implementations of Pthreads are written in C, then compatibility issues between libraries are eased.

The main part of the output from the generator phase is Fortran77 (this program is not intended to convert Fortran77 to C—other programs exist to do that), as is the code within each loop (which goes to make the body of other procedures). So the place for the C code to do its cunning work, is sandwiched in between. In fact we need two C routines for each loop, not just one. This is to handle the passing of variables. Figure 6.1 displays a graphical representation of the output program’s structure. To explain in more detail, the structure of the generated program becomes:

- The main (Fortran77) program remains basically the same as the input code.
- The code within each parallelised DO-loop is converted into a Fortran77 subroutine.
- For each loop that is parallelised, its entire body (everything from the DO-statement up to and including the line labelled as the end of the loop) is replaced with a single call to the first of the C routines.
**Fortran:**
Original code with all parallelised loops replaced with calls to C subroutines.

**C:**
- One top level C routine for each parallelised loop. This starts up a C-based thread for each iteration.
- One 2nd-level C routine for each iteration. Each calls the actual loop body.
- This is the actual Fortran loop body. It is extracted from the original loop.

Figure 6.1: *Diagrammatic representation of the calling-structure of the output program generated by the pilot system.*

- For each iteration of the loop, this C routine:
  - allocates memory (if required) for any local variables;
  - places all the relevant local and shared variables into a single customised data structure;
  - runs the `pthread_create` routine to spawn a new thread, passing a pointer to the customised data type through to the spawned function. This calls a second C routine.
• The second C routine is called and proceeds to:
  – de-reference the pointer passed to it;
  – unpack the customised structure to access all the variables;
  – call the Fortran subroutine which contains the actual code of the loop.

• The (Fortran77) routine containing the code from the loop in the original file runs for a single iteration before exiting. It makes changes to both local variables (which will be lost when this subroutine terminates) and to the globally shared variables (which will persist).

• When the Fortran iteration code completes, it exits, returning control within the thread to the second C routine which also exits.

• Having started a thread for each iteration, the top level C program then waits for each of the threads to complete (using the routine `pthread_attach`).

• When all threads have terminated, then the top level C routine exits returning control to the main Fortran program.

C has also been used as an “auxiliary” language for parallelisation of Fortran in several other packages. The output of SUIF (see section 1.1.1) is a combination of C and Fortran77. In the case of KAP (see section 1.1.8), no actual C code is output, but the code does make use of calls to a set of libraries written in C.

6.5 How it works

The code generator works by re-reading the input code on a line-by-line basis, writing to three output files as it goes. These files are the main (Fortran77) file, the auxiliary C file (containing both the C routines generated for each loop) and the auxiliary Fortran77 file, containing the actual code for each parallelised loop.

As the generator progresses through the original source file, it also traverses the list of records, which was produced by the analyser, containing
information about the various loops. All those lines outside parallelisable loops are just passed through into the main Fortran output file. When the generator reaches the start of a loop which has been flagged as parallelisable, then a more complicated process begins.

When processing a loop, the first thing that occurs is that all variables within it are analysed and broken into one of three categories: Local, Shared and Array. Local variables are all the scalars which are changed in the course of a loop iteration. Shared variables are all other scalar variables (ie. those which are only referenced). Any non-scalar are Arrays. A subroutine call is placed in the main output program. This is the call to the top level C routine for this loop (each loop has two unique C routines and one unique Fortran77 routine). The call passes through: all variables referenced or altered; all constants referenced; the loop variable (a special kind of local variable); the upper and lower bounds of the loop; and the step size. Since the label on the final line of the loop could conceivably be used by something else in the main program (for example an outer loop with the same termination), then a line with that label and a simple CONTINUE statement is inserted in the main file, just after the call.

Next, the other end of the call sequence is written, that is, the Fortran77 auxiliary file. This starts with a subroutine header and variable declarations. The variables are the loop variable and all Local, Shared and Array variables. Any constants referenced in the loop are also passed through in the same manner. That is, if $N$ is a constant integer with value 3 in the original file, then an integer variable named $N$ will be passed in with value 3. After the declarations, the code from within the loop is copied directly in as the body of the subroutine.

Now the second (in the calling sequence) of the C functions is generated. This function both accepts and returns a single value of type void*. The customised structure (for this loop) is defined. This structure contains exactly those variables passed into the Fortran77 loop code. The pointer is re-cast to be a pointer to the structure. Finally the Fortran77 loop subroutine is called with all the parameters from the structure being extracted and passed through individually.

The top level C routine (ie. the one which is directly called by the main program) is now coded. It starts with a header, to match the call in the main program as described above. Also declared, is the same structure
as described above for the second level C routine. Several local internal
variables are declared. Of note is the array of type pthread_t which will
store the thread identifiers (thread-ids are assigned uniquely to each thread).
Finally, after a call to pthread_init (which initialises some data structures
used internally by the Pthreads library), the main loop occurs.

The body of this loop is a large IF statement. It tests if the current
iteration is the last to be performed by the loop or not. This is important
because care must be taken in creating the multiple simultaneous copies of
the local variables. The main program may wish to access either the loop
variable or the Local variables after the loop has completed. At this point,
the standard dictates that the values must be those generated by the last
iteration. For example, in the case of the loop

\[
\text{DO 10 I}=1,N
\]
\[
X=\text{dble}(I)/\text{dble}(N)
\]
\[
10 \quad A(I)=X\times(X-1)/\text{sin}(X)
\]

any code which references \( I \) and \( X \) after the loop, should receive the values
from the last iteration. Namely, \( I=N \) and \( X=1 \).

Irrespective of the iteration, all the Shared and Array variables are
loaded into the customised structure. A new integer is malloced for the
loop counter, and is set to the correct value. This too is loaded into the
structure. If the loop is not on its final iteration, then space is malloced for
all Local variables. However if it is the final iteration, then the real vari-
ables (ie. those passed in from the main Fortran77 code) are loaded into the
structure instead. This means that in the final iteration, any changes made
to the Local variables persist beyond the end of the loop by being passed
back to the calling program.

At this point, the structure (in either case) is initialised correctly, and
a call is made to pthread_create, passing in the structure (cast as void*
as is required by the Pthreads standard). The identifier for each thread is
stored in the previously mentioned array of thread-ids. In this system, all
the threads (one for each iteration) are started as quickly as possible in one
batch. Once they have all been started, a second loop is run. Each iteration
of the second loop pthread_attaches to one thread of those started, and
waits for it to complete. When the last thread has finished, this top level C
routine exits, returning to the calling Fortran77 code.

6.6 Output of the generator

The product of the generator, and the end product of the entire system, consists of three files. These contain: a main Fortran77 program; Fortran77 subroutines corresponding to the code which was inside parallelised DO-loops; and C routines to link the main program and the subroutines in a manner which allows parallel execution. These three files are ready for compilation, linking and execution as a single, automatically parallelised program.

6.7 Limitations of the generator

There are only a three limitations in this section of the pilot system: the first concerns completeness, the other two relate to efficiency.

The pilot code generator is not complete, in that it does not support loops which run backwards. This was a deliberate omission for the sake of code clarity. The omission could be rectified by: changing the expression in the IF test; changing the size of the thread-ids array; and finally adjusting the loop conditions. All of these changes would take place solely within the top level C routine.

The first efficiency issue is that the generator, as it stands, quite freely mallocs memory for Local variables as well as for internal use, but never releases this memory into the free memory pool. This too, is trivial to fix, but has been omitted to avoid cluttering the illustrative code. All that is required to amend this deficiency is for the second loop in the top level C routine (the one which pthread attaches to each thread) to free all Local variables and the custom structure for each iteration. A free(Threads); should be appended to the very end of that routine (Threads is the array which stores thread-ids).

The second efficiency issue is a question of how the threads should be started, an issue first raised in section 2.2. The pilot system simply starts one thread for each iteration simultaneously. Having one thread per iteration,
all executing simultaneously, will give optimal load balancing. However it may not be ideal for performance. To begin with, running many threads (potentially millions) simultaneously may cause memory, cache or process-table problems. This can be overcome, whilst still maintaining the load balancing properties, by only starting $M$ threads (still with only one iteration each) initially. Then, when one thread completes, another can be started with a loop iteration which has not yet been started.

However in both of these cases, the thread start/stop overheads may become significant when occurring millions of times. If this is the case, an alternative is to arrange for each thread to perform several iterations, thus requiring less threads in total. If this is done, then the further question arises of how to allocate iterations to threads. The two most common techniques are “blocking” (where iterations 1, $\ldots$, $M$ are assigned to the first thread, iterations $M + 1, \ldots$, $2M$ to the next and so on) and “striping” (where iterations 1, $M + 1, 2M + 1, \ldots$ are assigned to the first thread; iterations 2, $M + 2, 2M + 2, \ldots$ to the next thread, and so on). As a general rule, striping is more likely to be balanced, but will probably make less effective use than blocking of any caches on the system.

### 6.8 Summary

The parser reads the original, serial Fortran source code and generates records with information about each line of code. The code analyser reads the information from the parser and generates information regarding `DO`-loops that can be parallelised. The code generator reads information from the analyser and uses it as a set of directives for generating Fortran subroutines, each of which can be executed multiple times in parallel. Each Fortran subroutine is comprised of the body of a `DO`-loop in the original code. The original `DO label . . . label` statements in the input code are replaced by a subroutine call.

The output from the code generator is three files: the Fortran77 main program; the Fortran77 subroutines; and the C routines which link them and actually perform the parallelisation. These three files are ready for compilation, linking and execution as a single, automatically parallelised program.
Chapter 7

Examples and timing results

The pilot system was tested using a number of different input programs, designed to test both the accuracy and the efficiency of the output. Obviously the most important criterion is accuracy. The program must be able to guarantee to generate code which is both correct, and equivalent to the input code. However efficiency is also a concern, since the entire reason for investigating parallelisation is to increase program speed.

7.1 Analysis demonstration

The following example program contains four loops, all of which are similar in nature to each other. However, of the four, only the last is parallelised by this pilot system. The program itself is meaningless, but is used to demonstrate some aspects of the system. The first loop is rejected since it contains a GOTO statement which references a label outside of the loop confines. If the two CONTINUE statements 10 and 20 are reversed, then the loop is parallelised. The second loop is in fact parallelisable, but (as mentioned in section 5.3) this pilot system refuses to parallelise loops with variable bounds (which this has). The third loop does have fixed bounds since Y is declared to be a constant 3. However it is genuinely not parallelisable due to data dependency in D. That is, the variable D is used before it is assigned a value. This means that the value of D used in the i-th iteration of the loop
is determined by the \((i - 1)\)th iteration. The final loop does not have this problem since \(D\) is assigned a value before it is used—the loop therefore can, and is finally, parallelised.

```fortran
Subroutine TestSub(B,T)
Parameter (Y=3)
Integer I,B(5),J,Y
Real X,S,A,T
Double Precision D
A=1
Do 20 I=1,100,2
   D=3*T
   Goto 10
   X=B(3)
   S=D
   D=J
20   Continue
10   Continue
   Do 40 I=1,100*A
      D=3*T
      X=B(3)
      S=D
      D=J
40   Continue
   Do 50 I=1,100*Y
      S=3*T
      X=B(3)
      S=D
      D=J
50   Continue
   Do 60 I=1,100*Y
      D=3*T
      X=B(3)
      S=D
      D=J
60   Continue
End
```

Figure 7.1: Sample program prior to parallelisation
When the sample program **TestSub** is run through the pilot system, three files are generated. The first is the top level file which is a replacement **TestSub** subroutine. The only change is that the final loop has been replaced with a call to P_LOOP_1 and a few comments. Note that neither J nor T are altered within the loop, so these can remain Shared. However X, D and S are all changed during the course of the loop, and so require local copies. All arrays (in this case just B(*)) are assumed to be Shared. Note also that the program has converted the 100*Y into 300, since Y is declared as a constant 3. This has been passed in as the upper loop bound.

```fortran
Subroutine TestSub(B,T)
Parameter (Y=3)
Integer I,B(5),J,Y
Real X,S,A,T
Double Precision D

A=1
Do 20 I=1,100,2
   D=3*T
   Goto 10
   X=B(3)
   S=D
   D=J

20   Continue
10   Continue
Do 40 I=1,100*A
   D=3*T
   X=B(3)
   S=D
   D=J

40   Continue
Do 50 I=1,100*Y
   S=3*T
   X=B(3)
   S=D
   D=J

50   Continue

C Parallel loop #1 begins
```

49
CALL P__LOOP_1(I, 1, 300, 1, B, J, T, S, X, D)

C Array referenced: B
C Locally declared: S
C Locally declared: X
C Locally declared: D
C Declared shared: J
C Declared shared: T

C Parallel loop #1 ends

60 CONTINUE

End

Figure 7.2: File 1: Top level code showing parallelism of the fourth loop (only) as a result of running the test program through the pilot system

The C routine P__LOOP_1, called above, is defined in the next file. This in turn starts the 300 threads, each running p__loop2_1_, which is also declared here. Note that the 300th iteration is started differently from the rest in that all variables which were deemed Local are passed directly in. This is in contrast to the other 299 iterations where disposable malloced space was passed in.

#include "pthread.h"
#include <math.h>

typedef struct { float r,i;} complex;

void* p__loop2_1_(void *input){

    struct my_type {
        int *loopvar;
        long int *B;
        long int *J;
        float *T;

    } m;


}
float *S;
float *X;
double *D;
};
struct my_type *my_var=(struct my_type*)input;
pf__loop_1_(my_var->loopvar , my_var->B , my_var->J, my_var->T , my_var->S, my_var->X, my_var->D);
}

p__loop_1_(int *realloopvar, int *first, int *second, int *step , long int *B , long int *J, float *T , float *S, float *X, double *D){
struct my_type {
int *loopvar;
long int *B;
long int *J;
float *T;
float *S;
float *X;
double *D;
};

int loopvar;

char *status;

int ThCount=0;

pthread_t *Threads=(pthread_t*)malloc(ceil((second-*first)/(*step)+1)*sizeof(pthread_t));

pthread_attr_t attr;
pthread_attr_init(&attr);
```c
pthread_attr_setscope(&attr,PTHREAD_SCOPE_SYSTEM);

for (loopvar=*first;loopvar<=*second;loopvar+=*step)
    if (loopvar<=*second-*step){
        struct my_type *my_var=(struct my_type*)
            malloc(sizeof(struct my_type));
        my_var->loopvar=(int*)malloc(sizeof(int));
        *(my_var->loopvar)=loopvar;

        my_var->B=B;
        my_var->J=J;
        my_var->T=T;

        my_var->S=(float*)malloc(sizeof(float));
        my_var->X=(float*)malloc(sizeof(float));
        my_var->D=(double*)malloc(sizeof(double));

        pthread_create(Threads+ThCount,&attr,
            p__loop2_1_,(void*)my_var);

        ThCount++;
    } else {
        struct my_type *my_var=(struct my_type*)
            malloc(sizeof(struct my_type));
        my_var->loopvar=(int*)malloc(sizeof(int));
        *(my_var->loopvar)=loopvar;

        my_var->B=B;
        my_var->J=J;
        my_var->T=T;
        my_var->S=S;
        my_var->X=X;
        my_var->D=D;

        pthread_create(Threads+ThCount,&attr,
```

52
p__loop2_1_,(void*)my_var);

    ThCount++;
    *realloopvar=loopvar;
}

for (loopvar=0;loopvar<ThCount;loopvar++)
    pthread_join(Threads[loopvar],(void**)(&status));

Figure 7.3: File 2: The C routines called in place of the loop

Finally, the actual body of the loop which was extracted from the input file has been written out to a third file within the new subroutine PF_LOOP1. This is called from p__loop2_1_ once by each of the 300 threads.

SUBROUTINE PF_LOOP1(I, B, J, T, S, X, D)

INTEGER I

INTEGER B(*)
INTEGER J
REAL T
REAL S
REAL X
DOUBLE PRECISION D

D=3*T
X=B(3)
S=D
D=J
60 Continue
RETURN
END

Figure 7.4: File 3: Fortran routine containing the actual body of the loop
7.2 Speed tests

This second test also uses a fairly meaningless piece of code. However it is useful here as the contents of the outermost loop are computationally significant for large values of $M$. The parameter $N$ controls the number of loop iterations. The input code is as follows:

```
Program SpeedTest
Integer N,M
Parameter(N=2,M=5000)
Double Precision x
Integer i,j,k
Do 20 i=1,N
   x=11.34
   Do 10 j=1,M
      Do 5 k=1,M
       x=i*(i-1)/(j*i+1)
      10 Continue
   20 Continue
End
```

Figure 7.5: Sample program to demonstrate efficiency

When this program is run through the pilot system, the outermost loop is (correctly) parallelised. The top-level code it generates is as follows:

```
Program SpeedTest
Integer N,M
Parameter(N=2,M=5000)
Double Precision x
Integer i,j,k

C Parallel loop #1 begins

CALL P__LOOP_1(i, 1, 2, 1, M, j, k, x)

C Locally declared: j
```
C Locally declared: k
C Locally declared: x
C Declared shared: M

C Parallel loop #1 ends

20      CONTINUE

end

Figure 7.6: File 1: Top level code showing parallelisation

The following is the other Fortran file created by the pilot system, it contains the contents of the loop which have extracted from the original source. The third (intermediate) file, containing the linking C routines was generated, but has not been included here.

SUBROUTINE PF_LOOP_1(i, M, j, k, x)

INTEGER i

INTEGER M
INTEGER j
INTEGER k
DOUBLE PRECISION x

x=11.34
Do 10 j=1,M
   Do 5 k=1,M
      x=i*(i-1)/(j*i+1)
   Continue
10 Continue
20 Continue
RETURN
END

Figure 7.7: File 3: File containing actual loop body

Both the original program, and the parallelised version generated by the pilot system were run for a number of different values of M and N. The
machine used in testing was a two processor Ultra-2 with Sparc processors, running SunOS 5.5.1. The following table contains the times (measured in seconds) from the tests. The column headed “Original” contains the time taken for the original program to run in serial for the given values of \( M \) and \( N \). The columns headed “Real” and “CPU” indicate the elapsed and cpu times (respectively) for running the parallelised version on the two processors.

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>Original</th>
<th>Real</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5000</td>
<td>23</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>5000</td>
<td>45</td>
<td>35</td>
<td>53</td>
</tr>
<tr>
<td>4</td>
<td>5000</td>
<td>93</td>
<td>57</td>
<td>104</td>
</tr>
<tr>
<td>8</td>
<td>5000</td>
<td>185</td>
<td>100</td>
<td>199</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>8.9</td>
<td>4.8</td>
<td>9.5</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>93</td>
<td>49</td>
<td>98</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>10</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 7.1: Comparison of results—unparallelised vs parallelised

### 7.2.1 Discussion of efficiency

From the above table, it can be seen that for two iterations or more (ie. for all values of \( M > 1 \)), the parallelised version is certainly faster in real time than the original. It can be seen that with \( M \) set to 1, the Real and CPU times are identical as only one thread is started, and so only one processor can be used. For all higher values, the Real time is less than the CPU time. When \( M \)—the number of iterations (and hence simultaneously running threads)—is greater than about 8, Real time is almost exactly one half of CPU time, the theoretical speed up is thus achieved. As can be seen, there is some overhead incurred by the threaded version since all the CPU times marginally exceed the times for the Original program. Judging from the numbers found in this trial, the parallelised program appears to be peaking at about 90% efficiency.
Chapter 8

Conclusions and future directions

8.1 Overview of achievements

This report has addressed a number of topics relevant to the automatic parallelisation of Fortran77 code. Many of these topics were explored by way of the discussion of the design and effectiveness of a pilot system written by the author. Two of the main objectives were complete automation of conversion from serial code to parallelised code, and portability of such parallelised code. Both of these were achieved.

Whilst not accepting all possible input, the pilot system (developed to show the practicality of the methods outlined in this report) has been largely successful. The system correctly identifies a number of parallelisation inhibitors. It also generates working, parallelised code for those loops it does deem parallelisable. This code is both portable, and developed in a fully automatic manner—no human interaction is required at all. Of greater importance: the code generated is provably equivalent to the given input code for several entire classes of input.

Finally, the generated programs achieves significant reductions in elapsed execution time on a shared-memory parallel machine. Testing has shown that it can achieve an efficiency of around 90% for some programs.
8.2 Future directions

The pilot system has three phases: the parser (see chapter 4), the analyser (see chapter 5) and the code generator (see chapter 6). The parser needs no significant design changes. It requires only the addition of some sections of the language specification which have been omitted to date. The code generator is even more complete, requiring only a few minor modifications to be completely general. This leaves the analyser, as the only section needing any significant work. Work is needed in two areas. One is support for analysis of data-dependencies within arrays. The other is a full code structure analysis (that is, tracing through IFs, etc. within the loops).

In addition to these functionality enhancements, the program can probably be improved to generate better output code from an efficiency point-of-view (see section 6.7). This would cause the generated program to be both faster, and to consume less memory.

8.3 Benefits and applications

Improvements in single processor technology are approaching a limit, and affordable, shared-memory, parallel machines are just now entering the market place. In addition, the unusually high degree of portability achieved by this system is largely due to the use of Pthreads. The relevant POSIX standard for this has only recently been accepted. Thus this study into automated, portable parallelisation is quite timely in nature as it would have been both more difficult, and less useful several years ago.

The pilot system is not yet completely developed in order to be used in a practical environment. However it does pave the way for a more comprehensive system which can also be used. Such a system, if it worked in a similar manner to this pilot system, would be a very powerful tool. Substantial reductions in program execution time could be gained without having to understand how a program works or how parallelism is achieved. All that would be required is the acquisition of the appropriate shared-memory parallel computer, and a copy of the parallelisation system.
There are many legacy computer applications, written in Fortran77 and having a high computational content, which would benefit from the application of an automatic code paralleliser, such as has been demonstrated.
Bibliography


[3] Banerjee, Utpal; Gelernter, David; Nicolau, Alex and Padua, David (Eds.), *Languages and compilers for parallel computing*, Lecture notes in computer science, (768), Springer-Verlag, USA, 1993.


[5] Goff, Gina; Kennedy, Ken and Tseng Chau-Wen, *Practical dependence testing*, Department of Computer Science, Rice University, USA.


[14] Bodin, Francois; Beckman, Peter; Gannon, Dennis; Gotwals, Jacob; Narayana, Srinivas; Srinvas, Suresh and Winnicka, Beata, *Sage++: an object-oriented toolkit and class library for building fortran and C++ restructuring tools*, OONSKI ’94, in press.


Acknowledgments

I would like to acknowledge and thank my supervisor, Dr George Havas, for the insights and practical assistance given so freely during the course of this project. Thanks is also due to the Physical Sciences and Engineering Research Computer Facility, for their kind provision of computer resources.