UNIVERSITY OF SOUTHAMPTON

Faculty of Engineering and Applied Science
Department of Electronics and Computer Science
Southampton SO17 1BJ

TOOLS FOR THE SUPPORT OF AUTOMATIC PARALLELISATION BY ALGORITHM SUBSTITUTION

by Philip Galloway
BSc

A thesis submitted in partial fulfilment of the requirements for the award of a degree of Master of Philosophy at the University of Southampton.

Department of Electronics and Computer Science

April 2002

SUPERVISOR: Dr J S Reeve
Department of Electronics and Computer Science
University of Southampton
Southampton S017 1BJ

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

Faculty of Engineering and Applied Science
Department of Electronics and Computer Science

Master of Philosophy

TOOLS FOR THE SUPPORT OF AUTOMATIC PARALLELISATION BY ALGORITHM SUBSTITUTION

By Philip Galloway
BSc

Program recognition is an important tool for the development, support and manipulation of software, particularly in the field of parallelisation tools. One area of current interest is the potential advantage of algorithm substitution as a means of optimising source code for execution on parallel hardware. In order to substitute an algorithm within a source code, the elements of the original algorithm must be recognised and extracted. The recognition of code elements is therefore the first step towards implementing such a system.

This work reports on the implementation and operation of two algorithms to support the program recognition aspects of this automatic algorithm substitution idea. The algorithms have been applied to a number of source code examples and their performance evaluated. The approach embodied in these algorithms has been shown to be effective as they can operate without the need for extensive code normalisation found in other approaches to this problem.

Acknowledgements

I would like to thank Dr Steve Simpson for his unwavering support and encouragement throughout the duration of this work. I would like to thank Roke Manor Research Ltd for their financial assistance in allowing me to pursue this activity as a part time research student.

My thanks also go to Dr Jeffery Reeve who has provided guidance and encouragement during my work in the ECS department.

I would like to dedicate this work to my Mother.

Contents

Chapter 1 Introduction 1

1.1 Background 3

1.1.1 Problem domains for parallel processing 5

1.1.2 Support for the programmer 7

1.1.3 Parallelisation Toolsets 8

1.1.4 Current Parallel tools and technology 11

1.2 Motivation 17

1.3 Related Work 18

1.3.1 Automatic Program Recognition 19

1.3.2 Algorithm Substitution 19

1.3.3 Algorithm Learning Procedures 20

1.4 Approach 21

Chapter 2 Learn Tool System Description 24

2.1 Learn Tool History and Composition 25

2.2 The Sage++ System 26

2.3 Reverse Traversal 26

2.4 Pattern Matching 29
2.4.1 Related Work on Pattern Matching 29
2.4.2 Overview of Approach 30
2.4.3 Code Status Items 32
2.4.4 Rules 33
2.4.5 Keys 33
2.4.6 Matching Algorithm 33
2.4.7 Learn Tool Facilities 35

Chapter 3 Test Cases 38
3.1 Statistics Example 39
3.1.1 Reverse Traverse 39
3.1.2 Pattern Matching 42
3.2 Laplace Example 47
3.2.1 Reverse Traverse 48
3.2.2 Pattern Matching 49
3.2.3 Laplace Variation 1 52
3.2.4 Laplace Variation 2 53
3.2.5 Laplace Variation 3 53
3.2.6 Laplace Variation 4 53
3.3 Matrix Multiply Example 55
3.4 Maximum Value in an Array of Integers 56
3.5 Discussion 56

Chapter 4 Concluding Remarks 58
4.1 Parallelisation Tools 59
4.2 Reverse Traversal 59
4.3 Pattern Matching 59
4.4 Summary 60

Appendix 1 – Example Source Code 62
A1.1 – Statistics 62
A1.2 – Laplace 67

Bibliography 73

List of Figures

Figure 1 Generic Parallelisation Toolset components 8

Figure 2 Abstract view of program composition 19

Figure 3 Reverse Traverse Flow Chart 27

Figure 4 Flow chart detailing the rule-matching algorithm 35

Figure 5 Annotated Rule Editor Dialog 36

Figure 6 Annotated Rule Association Dialog 37

Figure 7 Screen shot of Stats example reverse traverse showing mean dependency graph. 39

Figure 8 Reverse Traverse dependence graph for variance. 41

Figure 9 Rule: calculation of mean 42

Figure 10 Screen shot of the match detail for the calculation of mean rule. 43

Figure 11 Rule: sum array items 44

Figure 12 Screen shot of the match detail for the Sum array items rule 45

Figure 13 Rule: Assign ToZero 45

Figure 14 Screen shot of the match detail for the AssignToZero rule 46

Figure 15 Rule: Accumulator_1 46

Figure 16 Screen shot of the match detail for the Accumulator_1 rule 47

Figure 17 Reverse Traverse dependence graph for data1 48
Figure 18 Rule: laplace sequence 49
Figure 19 Rule: ew then ns 49
Figure 20 Rule: west then east 49
Figure 21 Rule: 2D array West access 50
Figure 22 Screen shot of the match detail for the "laplace sequence" rule 51
Figure 23 Screen shot of the match detail for the 2D array West access rule. 51
Figure 24 Variation 1 coding of Laplace example 52
Figure 25 Screen shot of the match detail for the" laplace sequence" rule for this variation 52
Figure 26 Variation 2 coding of Laplace example 53
Figure 27 Variation 3 coding of Laplace example 53
Figure 28 Variation 4 coding of Laplace example 54
Figure 29 Screen shot of the match detail for the "laplace sequence" rule for this variation. 54
Figure 30 Code fragment for matrix multiply 55
Figure 31 Code fragment for find maximum integer value from array. 56

List of Tables

Table 1 Description of the fields in a "CodeStatusItem" 32

Chapter 1 Introduction

> "Automated program recognition can play a crucial role in overcoming limitations of existing tools for automatic parallelization" Martino et al. [1].

Program recognition is a process whereby the core functionality of an application, module or phrase is recognised by studying its implementation or detailed behaviour. It can be thought of as the process whereby the original intent of the programmer is deduced. Indeed the working title of this thesis was "Intent Analysis", however program recognition is the more commonly used term in the computer science community.

Software engineers who maintain legacy code perform program recognition routinely, as do application developers who work in teams and need to review the work of their co-workers. The application of program recognition to the process of parallelisation builds on the observation that the greatest program speed-ups are most often achieved when a component algorithm is completely changed rather than re-coded or optimised. This is particularly true for algorithms that are targeted for parallel hardware where the memory hierarchy is many layered and inter-process / inter-processor communication costs can dramatically inhibit the efficiency of poorly matched algorithms. The other application of program recognition for parallelisation activities is to restructure the code to use pre-optimised library routines once a suitable substitution has been found.

Automated program recognition would be used to allow a parallelisation tool to perform algorithm substitution as part of its operation. Algorithm substitution may also be useful for optimising code for serial platforms, indeed for commercial software that is developed under a tight budget, optimal algorithms are rarely chosen for an initial implementation. There are several reasons for this, the most common being the drive to implement a simple (less efficient) algorithm accurately in the minimum required development time so that effort can be concentrated on verification of functionality and user interaction issues.

Several workers in the academic community have reported on their progress towards the goal of implementing algorithm substitution systems, these include Martino et al. [1] Keßler et al. [2], Pinter at al. [3], Keßler [4,5], (Raghavendra et al. [6], Bansali et al. [7] and Hagemeister et al. [8]. This work was motivated by the same fundamental idea and has resulted in a prototype analysis code, named "Learn Tool" that implements two of the strategies considered.

This work seeks to allow the deduction of the intent of the programmer to be partially automated, to assist in the optimisation of legacy code for operation on parallel platforms. Thus the scope of the work is confined to the extraction of intent information from legacy code and does not address directly the issue of optimal program fabrication for parallel platforms.

The thesis is organised as follows:

Chapter one starts with a background section that discusses briefly the distinction between parallel and serial computers and then sets the context of the work with a description of a generic parallelisation toolset that may be used to convert serial code to execute "optimally" on parallel hardware. This is followed by a more detailed discussion of the motivation for the work. Next, a section detailing the related work on program recognition provides an overview of other work in this area and how it relates to this work. The chapter concludes with an outline of the approach to the practical work undertaken.

Chapter 2 describes the software constructed to investigate the ideas for the program recognition concept; this introduces the Sage++ toolset (a third party parse tree generator and source code to source code transformation toolset) the Reverse traversal and the Pattern matching algorithms, which were implemented on top of Sage++ in the Learn Tool application.

Chapter 3 provides examples of exercising the Reverse Traverse and Pattern matching algorithms on simple source code test cases. This highlights some of the difficulties of the two approaches and demonstrates the basic utility of the pattern matching approach.

Chapter 4 contains the conclusions drawn from the work and summarises the main findings.

1.1 Background

To start it is worth reviewing the definitions of parallel and serial computer, as in these days of high integration and pipelined processors the distinction can become somewhat blurred.

A serial computer can be thought of as executing instructions in an ordered manner, starting with the first and ending with the last, essentially adopting a Von Neumann architecture. The order of instructions is determined by the control structures in the program but there is no simultaneous execution of different parts of the instruction sequence at any time. A parallel computer has more than one processing unit allowing separate instructions to be executed simultaneously on different units. The instructions may be the same but operating on different data, Single Instruction

Multiple Data (SIMD) or both instructions and data may be different, Multiple Instruction Multiple Data (MIMD). SIMD is often referred to as a data parallel approach or as the parallelisation strategy is almost always based on data partitioning. MIMD is also described as task, thread or process parallelism.

Parallel implementations of software operate in two main paradigms. The first is closely associated with SIMD or the data parallel approach. The partitioning of the data in the problem is embedded in the source code, typically in HPF or OpenMP extensions to C, Fortran or C++. The distribution of the data to physical processors, or processes on actual processors is a compile or run time issue and is dealt with by the compiler and/or a data distribution library. The second, MIMD style is a lower level approach based on writing explicit message passing code and potentially running different executables on the distributed processors. The second style can be used for a SIMD program and can provide a more intimate control of the locality of the data. The second style is the most common approach for implementing task parallel applications. The two main task parallel strategies are master/slave and pipeline. The master/slave model is were a single co-ordinating process distributes tasks to a pool of slave processors that may perform multiple instances of the same code or may perform different tasks associated with the overall problem. A pipeline strategy breaks up the calculation into stages that feed from one process to the next with data flowing through them, very much like a production line. Each processor performs part of the overall calculation and the data is passed on to the next processor when processed. Pipeline parallelism is often employed in Digital Signal Processing (DSP) situations where there is a natural flow of data from an external source and 'live' processing is required.

Modern Microprocessors are parallel processors, in that they typically have multiple processing components, Central Processing Unit (CPU), Floating Point Units (FPU's) and dedicated communication sub processors, along with memory management and cache facilities. For the purposes of this text, microprocessors of this sort will be classified as serial machines, since the instruction set (for the high level programmer) is purely sequential, with any parallelism being exploited invisibly by the compiler, macro code, or skilled assembly language programmer.

Modern compilers achieve a good degree of efficiency for most applications where there is fine-grained parallelism. This granularity maps well onto the multiple processing units within typical microprocessors.

Parallel computers exist in many configurations ranging from dual processor PCs, and workstation clusters to dedicated parallel machines such as IBM SP2, Cray T3D, T3E and SGI Origin 2000. Every parallel computer has a set of performance characteristics that relate to its memory hierarchy, inter-processor communications and core microprocessor. The programming model for a parallel machine may be shared memory, distributed memory or both. The level of hardware support for communications will influence the programming model and how effectively it can be used on a particular platform. For example, a network cluster of workstations on a standard Ethernet, will have high latency and relatively low bandwidth communications, so a shared memory paradigm would tend to perform poorly against, for example an SGI Origin 2000, which whilst having a physically distributed memory, has high bandwidth support for node to node communications (as well as a common virtual memory space). That is not to say that programmers cannot use a shared memory paradigm on a parallel machine with low communications performance, but rather to indicate that that it will only work efficiently if great care is taken to ensure data and process locality, a task, often more easily achieved by adopting a message passing approach.

Parallel machines roughly fall into three categories, Symmetric Multi Processing (SMP) where a single memory bus is used for all processors, Massively Parallel Processing (MPP) where the memory is distributed and data is shared by communications and recently Non-Uniform memory Access (NUMA) or Distributed Shared Memory, (Origin 2000) where there is explicit hardware support for a single memory image over a distributed memory architecture.

1.1.1 Problem domains for parallel processing

Parallel processing is adopted as a solution to the problems of ultimate computer performance and cost performance trade-offs Kelly [9]. Parallel architectures offer the possibility of exceeding the processing capability of single processors in terms of MIPS and FLOPS as well as ability to provide large memory configurations with

distributed address spaces. Many areas of research can and do utilise the fastest of computers to tackle problems in domains such as weather forecasting Sabot et al [10], Fluid Mechanics, Sethian [11], Jin et al [12], Quantum Physics, Hong [13], Wave Propagation, Ewing [14], Computational Mechanics, Cross et al. [15] and my own area of interest, Electromagnetic Simulation, Harman and Simpson [16], Galloway and Simpson [17], Galloway [18] to name a few.

To summarise, parallel computers are used in the following situations:

- Where a single processor reaches its ultimate speed and is still not fast enough.
- Where sufficient execution speed can be attained on a single processor but the cost of the equivalent parallel hardware is lower.
- Where a single processor configuration cannot address enough memory space for the calculation of very large problems.
- Where a convenient parallel resource, such as a workstation cluster has available idle time and can be exploited at low cost. This is common in most business environments where hundreds of desktop PC could be exploited at little additional cost and with little impact on the main users.

The use of parallel processing machines in other situations (other than parallel processing research) is rarely cost effective, since the operator has to overcome all the drawbacks of operating a parallel application listed by Kelly [9] which do not occur for the alternative realisable single processor solutions.

The above holds true for most scientific applications, whereas there is considerable activity in internet based parallel processing, where agents, Knapik and Johnson [19] may interact to achieve goals that are not possible on a local configuration because of special resource or information availability. Languages to support this work such as Java 2 (JDK 1.2), Oz and Mozart, Haridi, Roy et al. [20] are not considered in the context of this work, which focuses on scientific applications and primarily legacy FORTRAN applications.

1.1.2 Support for the programmer

A programmer writing code for a parallel computer is operating in one of the following situations:

- Converting an existing serial application.
- Converting a parallel application to a new computer platform.
- Writing a new application for a parallel computer.

The conversion of existing code entails a trade-off between changing inefficient code sections and spending a minimum of effort in performing the conversion. Parallelisation tools have been developed to attempt to make these tasks efficient. The tools provide a variety of facilities that help the programmer solve the problems encountered during the code creation and conversion processes.

The main problems that the programmer has to solve are:

- How to distribute or partition the data.
- How to distribute or partition the functionality.
- How to take account of the memory layout of the target machine or machines.
- Making the selection of suitable algorithms.

Incidental details of how to implement the data sharing and synchronise the operation of multiple processing units are also areas where tools can assist.

With no parallelisation tools it is possible to construct parallel applications using networking protocols such as TCP/IP, which are usually accessible from most high level languages, however for any particular application the first step is most likely to build a communications library or a virtual memory space support structure on top of this layer. Commercial Off The Shelf (COTS) parallelisation tools provide these with minimum effort and go on to provide support for code analysis, code re-writing and supporting debugging and monitoring tasks.

1.1.3 Parallelisation Toolsets

The practical work undertaken for this thesis looks at two components of a generic parallelisation toolset. This section presents a representative parallelisation toolset and highlights where the work reported on in this thesis resides in the overall parallelisation process.

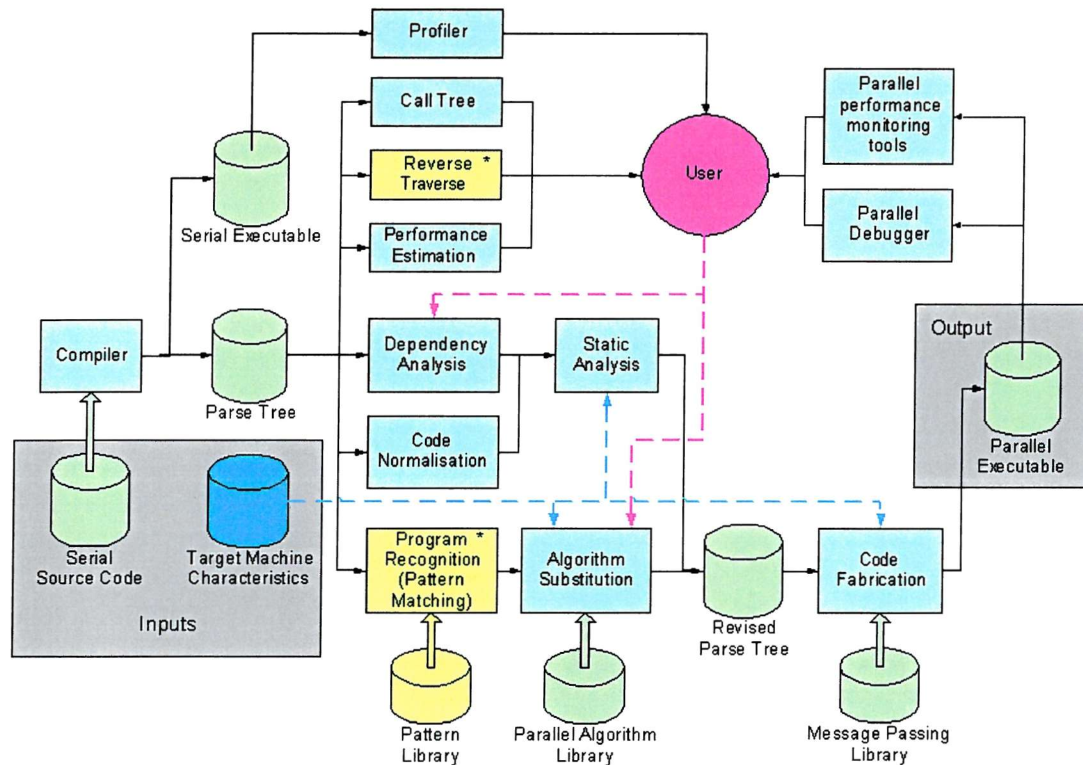


Figure 1 Generic Parallelisation Toolset components

Figure 1 provides a detailed diagram showing some of the key components of a generic parallelisation toolset. The main purpose of the toolset is to assist the user in converting a serial application to run effectively on a particular target parallel machine. The application source code and the target machine characteristics (along with any target machine specific libraries) constitute the starting position of the conversion process. The toolset provides the user with facilities to understand the code, manipulate the code and evaluate the efficiency and correctness of the resulting parallel executable. The output is the **parallel executable** (or several executables) that will operate on the target parallel machine. This toolset is assumed to operate on a traditional high level language such as FORTRAN, C or C++.

The interconnections in Figure 1 show the general flow of activity and highlight where machine characteristics and human support or influence is needed or expected. Note the practical use of the toolset would be iterative, with the user potentially revisiting earlier stages of the parallelisation process several times to fine tune and optimise the end result. The remainder of this section details each of the components on the diagram and provides a brief explanation of the support being provided by the toolset.

The **Target machine characteristics** would include details of the number of processing nodes, the memory configuration, the cache line lengths and communication performance such as latency and bandwidth. These details are often needed to optimise data partitioning, algorithm selection and the code fabrication processes.

The compiler would produce a serial executable and a parse tree, which may be used by the other tools in the toolset to manipulate and extract information from the code.

In a large application it is often only a small section of code that can provide a significant benefit from parallelisation. This is typically in a deeply nested loop where most of the application spends its time during the calculation. Identification of this code section is critical to effective parallelisation. The toolset would provide a **profiler,** which would instrument the code to identify the main time consuming operations that the application performs. Alternative means of identification of the key section to parallelise could be provided by a **call tree**, a **performance estimation** tool or perhaps a **reverse traverse*** tool. In most cases all these would be used to identify the critical sections of code and to put into context the sections with respect to the overall architecture of the application.

Dependency analysis is the key process that allows the identification of independent processing opportunities. The loops in the key sections of the code are examined and the references to the data being processed is checked to see if the operations can be performed in parallel or if the operations must proceed in a strict order Wolfe

---

* One of the tools investigated in the practical work of this thesis

[21][22]. Prior to dependency analysis the code may be manipulated to make it more likely for the dependency analyser to operate effectively. These manipulations are often called **code normalisation** and may consist of but not limited to:

- Procedure in-lining.
- Forward propagation of constant expressions.
- Induction variable substitution.
- Temporary variable substitution.
- Dead code removal.
- Conversion of GOTO's into if-then-else or while statements.
- Loop distribution.

Other manipulations such as loop unrolling and loop re-rolling may also be used to improve the outcome of the dependency analysis. Some toolsets may provide facilities for the user to influence the outcome by feeding in user knowledge of the application to allow more data independence to be identified.

Static analysis is a process whereby a decomposition of the program data to execute in parallel is selected. This results in a SIMD code with the data partitioned among the processors and the application of an owner computes rule. A classic example of this occurs for structured grid algorithms, where there may exist several data partitioning strategies BLOCK, STRIP, CYCLIC etc. which can be selected at the static analysis stage. In cases where the decomposition is likely to be data dependent, for example when the data indexing is indirect, then this might lead to the addition of code to perform a dynamic decomposition, which is then evaluated at runtime.

The **code fabrication** operates on the revised parse tree and assembles and compiles the parallel executable(s). This may consist of adding message passing code to distribute and re-assemble the parallel sections of data processing, or may entail the addition of OpenMP or perhaps HPF directives. Again this tool may need to be sensitive to the target machine properties, for example selection of data sizes may

crucially effect the cache coherency, also the possibility to pre-fetch or hide communication operations behind processing may be possible for some architectures, Brooks and Warren [23]. Vectorisation of communications to minimise the impact of communications latency may also be provided.

Program recognition* may be used to automatically recognise the mathematical operations embodied in an application and provide the possibility to substitute algorithms that are better matched or are more parallelisable on the target machine architecture. The process of program recognition is the main subject of this thesis and usually involves template or pattern matching. **Algorithm substitution** is the process whereby an equivalent algorithm is evaluated for use in a particular application, as a substitute to one of the recognised components within the original code. This selection involves an assessment of the likely performance of the new algorithm and verifying that a complete substitution is possible. Code normalisation may also precede program recognition.

The final components for a parallelisation toolset are a **parallel debugger** and **performance visualisation** tools. These allow the correctness of the parallel code to be tested, corrected and potentially fine-tuned.

1.1.4 Current Parallel tools and technology

Several of the components for the generic parallelisation toolset are already well established in the market place, whilst others are still being developed and improved. This section provides a summary of where this field was at the start of the practical work undertaken for this thesis and then highlights some of the advances made in the interim.

Parallel Languages

Parallel languages allow the direct exploitation of any possible parallel nature in an application. A typical application will have tasks that can be done in any order and some which must be executed sequentially. The programmer can code the sequential

* One of the tools investigated in the practical work of this thesis.

parts of the task and then indicate which ones can be performed in parallel by using special language constructs.

There are several disadvantages to the parallel language approach:

- Existing applications have to be completely re-written
- The languages are sometimes only available on a small number of specialist machines
- The application is not generally portable
- Only a few programmers specialise in the new languages
- The languages may not support sophisticated features of the more general-purpose languages.

Occam, Poutain and May[24] and Strand, Foster and Taylor [25] are two such languages that have gained some acceptance in the community despite suffering the drawbacks mentioned above. Parallel languages do not appear as part of the generic parallelisation toolset as this is aimed at converting existing code and not re-writing from scratch. Having said this, if automatic translation into a parallel language were possible this might be an appropriate starting point for parallelisation.

Language Extensions

Extensions to commonly used languages such as FORTRAN 77 allow certain operations to be performed in parallel. Large matrix and array operations are typical. The extension are either of academic origin or from parallel hardware vendors such as Silicon Graphics Inc. Instances of these language variants/extensions include Vienna FORTRAN, Chapman et al [26], FORTRAN D, Fox et al.[27], HPF, Harris et al. [28], ADAPTOR HPF, Brandes et al. [29], OpenMP [30], Jin et al. [31], MIPSpro™ Power Fortran 77 [32], CM Fortran [33] PARADIGM Su, Lain et al.[34]. The programmer suggests a distribution scheme for the data, typically a mesh partitioning using compiler directives or the extended language syntax and then performs loop operations on the distributed data in parallel. The compiler checks that data is passed correctly across the partitions and may provide a set of alternate

distribution strategies (CYCLIC or BLOCK). These language extensions can be very useful for building new parallel code, and in certain circumstances adapting existing applications.

The main drawbacks appear to be the limited problem domains where these operations are beneficial. The programmer still needs to know a lot about the algorithm and the expected flow of data to select an appropriate partitioning. Conversion of a code containing indirect or conditional addressing of arrays is a serious difficulty as highlighted by Walker, [35] although some solutions have been suggested for these problems, Das and Slatz et al., [36]. More recently there has been much activity on sparse systems and there exist a number of parallel libraries to address these problems. Wijngart [37] provides an extensive list of the current work in this area as part of the conceptual design paper on the Charon toolkit, for example Saad et al [38] with PSPARSLIB and Schonauer et al with LINSOL [39].

The portability of HPF makes it an attractive language for developers indeed the extensions in HPF2.0 allowing task parallelism to be addressed make this an extremely versatile language.

Message Passing Libraries

Message passing libraries provide a machine independent communications facility for message passing between independent processes. The processes are independent instruction sequences being executed on separate processors, or time-sharing one processor. These libraries are typically implemented on top of TCP/IP sockets, pipes or streams, dedicated IO links, parallel data interconnects on shared bus systems or shared memory areas. The most commonly used library is called the Message Passing Interface (MPI) [40] [41] although different vendors of parallel machines previously generated similar but incompatible products, these include CsTools from Meiko [42], Express, PARMACS plus similar offerings from SGI.

These very basic components allow the parallel programmer to concentrate on the problem and ignore some of the details of the communication protocols. The topology independent features are both an advantage in that the programmer can completely ignore the topology issue, and a disadvantage in that unfortunate

scheduling of task locations and data partitions may lead to a failure to attain the best efficiency.

Decomposition Assistance

Many mesh problems in science have a very obvious set of partitioning options. A cubic grid for which all cells require the same processing load, and nearest neighbour communications, is generally split in one dimension and partitioned into equal slices. Two or three-dimensional slicing is possible, but this is not generally useful for exploiting a computer with a non-square or cubic number of processing nodes. Many physical problems have more complex grids, fluid dynamics models of rivers for example are only active where the fluid is to be represented. The connectivity and distribution of the data points in the computer memory is sometimes a sparse matrix, with only a subset of the elements active. Partitioning of sparse matrix systems is a classical graph problem. The main aim is to level the number of nodes in each partition whilst minimising the amount of data required to be communicated along the edges. Saad and Sosonkina [43] provide a good overview of the general approach for parallelisation of a sparse linear system. The domain Decomposition Tool DDT, Flores and Reeve [44] is a typical partitioning program that locates good partitions by a number of methods. It is important to realise that the best partition must be a good sub optimal partition that can be located with minimal computation effort, since the runtime of the complete problem must allow for the effort consumed in selecting the partitioning. Generally the optimum partition will take much too long to locate. Optimum partitions share the load on the processors evenly and minimise the communications at data partition boundaries.

In unstructured grid or sparse matrix problems communications along the edges of the partitions become very complex and vectorisation of communications become difficult. In recent years much of the focus of work has been in these areas where parallelisation is non trivial. Examples of this include the work of Brandes [29] and Keßler [5], Adams[45], Saad and Sosonkin [43].

Code Analysis

These tools are aimed at re-engineering existing serial applications. In many cases the original programmer is not involved in this task, the code may be poorly documented, badly structured and may be prematurely optimised Symonyi, [46]. The programmer faces the task of understanding what the code is doing, how it is doing it, and finally how it can map onto a parallel platform.

Tools to aide in this process contain a front end parsing for the language and then instead of the code generation functions of a compiler, they generate dependency or data logistics information that can be browsed by the user. FORGE Explorer [47] and IDA, Merlin [48] are typical of these. The Sage Toolset [49,50] is available to academics for building analysis and code restructuring tools (a machine tool toolset). This tool provides a front end parser and a C++ class library that allows access to the data contained in the source code.

Commercial systems such as KAP™ from Kuck and Associates Inc. are now becoming standard tools for the support of paralleisation / optimisation. The Digital KAP Fortran / OpenMP optimiser includes support for:

- Automatic and directed parallel decomposition for SMP
- Loop optimisations
- Memory Management optimisations
- Scalar optimisations
- Function in-lining
- BLAS recognition
- Dusty Deck transformations
- Informational program listings.

KAP [51] provides Inter procedural analysis IPA as well as code transformations. It has a limited capability to provide algorithm replacement with calls the Basic Linear

Algebra Subroutines (BLAS) libraries, some of which are optimised for target parallel architectures. The Charon toolkit, Wijngaart [37] is targeted at structured grid problems.

OpenMP [30] supports multi-platform shared memory architectures, where each processing node shares at least some of its memory space with all the other processors taking part in the application. This common memory space model can support both MIMD and SIMD programming styles, although there can be serious performance penalties if it is applied to MPP systems and includes fine-grained parallelism.

Code Restructuring

The KAP/Pro Toolset, Kuck and Associates, [51], IBM PTRAN [52], SUPERB Zima et al. [53], CMAX Sabot et al. (Connection machine) [10], CAPTools, Cross et al. [54] [15] and PARAMAT, Keßler [4] are instances of code generation tools that attempt to automate the whole process of converting a serial application for optimum performance on a parallel computer. These automatic parallelisation tools all adopt the technique of examining source code to identify independent calculation threads, and generating new code that distributes the operations and or data over the nodes of a parallel machine, this process is usually termed dependence analysis and is described in detail by Wolf [21,22]. When the parallelisation systems are well refined, they can manage to achieve a reasonable speed up. Good performance improvements are most easily achieved on naturally data parallel algorithms Cross, Ierotheou et al., [54]. Cross et al. CAPTools [12][15] is a toolset that originated for structured mesh problems and appears well suited to the parallelisation of CFD codes. There exist other code restructuring tools such as the Foresys Fortran Engineering System from Simulog [55] that are more suited to code normalisation, porting and code maintenance.

Parallel Debugging and Runtime Analysis

At the back end of a parallelisation task the resulting parallel program will be tested. This is often when the performance bottlenecks are discovered, Galloway [17]. In cases where performance is less than expected, it is useful to be able to observe the

program in action. For this task, parallel performance analysis (PPA) tools may be used to gain useful information on the runtime behaviour of the program. Parallel debugger support may also be required if the parallelisation process has introduced errors in the algorithm. These tools become increasingly important as the complexity of the parallel program increases.

A sample of tools of this sort include the P2D2 project, Hood [56] which is aimed at the CFD community to provide a consistent parallel debugging environment across multiple architectures, an essential feature if the parallelisation is on a heterogeneous platform or platforms. It is based around gdb and has been exercised on IBS SP's, SGI worksations and Origins, and on Linux systems. Panorama, May and Berman [57] is a parallel debugger and performance tool based on trace collection. ArrayTracer Nikolaou et al. [58] concentrates on performance analysis and attempts to minimise the impact of trace collection using a sophisticated static analysis prior to runtime. PAPI, London K., Dongarra et al, [59] provides performance monitoring using hardware counters for parallel applications on Linux platforms. The combination of the profiling support in the Tuning and Analysis Utilities (TAU) toolset and the runtime interaction from the Distributed Array Query and Visualization Framework (DAQV) Shende et al. [60], provides a different analysis view based on callstack sampling.

1.2 Motivation

A means of being able to recognise core algorithms and be able to replace them with efficient alternatives for the particular target hardware would seem to be an ideal way of providing a more complete code restructuring tool for the parallelisation of dusty deck code. I will call this process "automatic algorithm substitution" if this is performed automatically by a code-restructuring tool.

This term "automatic algorithm substitution" has been used but is as yet not clearly defined. From a mathematical viewpoint for procedures within a source code there are many methods of implementing the same required algorithmic functionality. For example the number of sorting algorithms developed in the programming community is large: Bubble Sort, Quick Sort, Selection Sort are but a few, an interesting account of a few of these algorithms is provided by Meader [61]. The choice of the

algorithms for a specific program is influenced by many factors, knowledge (of candidate algorithms), target machine characteristics, speed requirements etc. In many cases, as a program ages these factors change dramatically, particularly if an implementation on a parallel machine is required. At this stage the choice of a different algorithm for part of the program may lead to a significant optimisation. Intent Programming (IP), Symonyi [46] builds on this recognition that as the environment changes, the ideal source code for an application also needs the ability to change to preserve efficiency.

For new application generation there has emerged a trend favouring the adoption of several levels of abstract design before code is written. Fitzpatrick et al. [62] contends that

"a competent mathematician can write functional specification in a few hours"

that can subsequently be transformed to optimal code by utilising a library of proven optimal transformations. Cate et al. [63] describes the experience of porting applications to parallel machines and promotes the concept that all arbitrary implementation specific design decisions need to be documented as the code is built to allow efficient re-engineering for a new parallel or serial computer. His paper cites a specific instance where the computer architecture determines the optimum choice of algorithm selection. If the compiling system cannot change algorithms, then it will often fail to generate efficient code for parallel computers.

The goal therefore is to provide a means of re-fabricating programs to use algorithms that are well matched (efficient) to a target platform. The recognition of algorithmic content is the first step to achieve this goal. The identification of algorithmic content of a dusty deck program may be thought of as an inverse problem to that of compiling. It is a translation from the specific to the general and is a challenging problem for a computer based tool.

1.3 Related Work

Having set the scene in the previous section, this section talks in detail about the directly related work on algorithm recognition as a means of providing improved optimisations for applications running on parallel hardware.

1.3.1 Automatic Program Recognition

Early work on designing systems for automatically recognising program content seems to have started in the late 1970s mainly in the artificial intelligence community for example Waters [64] and Fickas et al. [65]. Their work provides the use of the term clichés to designate commonly recurring sequences of standard low-level instructions and control sequences. A typical approach to the recognition of these clichés or phrases is by pattern matching on the program text, parse tree or data flow graphs. GRASPR Willis, [66] is an experimental system that adopts the latter approach with some success. The reported motivation for GRAPSR was for assisting software engineers as well as addressing an interesting artificial intelligence problem. Examples cited for the use of the technique range from automatic documentation generation too code maintenance and reverse engineering. Alternative terminology for this assembly of low level instructions varies with author and ranges from 'semantic concepts' Kozaczynski et al. [67], too 'plans' Rich [68] and 'idioms' Pinter et al. [3] and Snyder [69]. Algorithm substitution for parallel program optimisation seems to have been recognised as a possibility much later on.

1.3.2 Algorithm Substitution

A programmer deals with code on many levels of abstraction. One way of looking at this would be to say that programs are collections of algorithms glued together with interfaces. The algorithms are made up from simple component functions, which in turn are built up from basic operators. Figure 2 illustrates this conceptual outline.

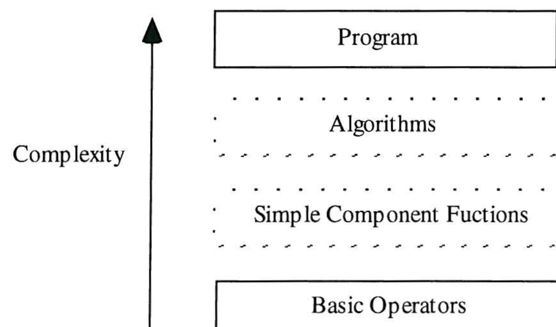


Figure 2 Abstract view of program composition

For program recognition, as described by Wills [66] the clichés would be the simple component functions and/or whole component algorithms.

Cate [63] and Simonyi [46] clearly emphasise that the choice of algorithm should be one of the last operations in the program design process and in an ideal environment would be an easily changed design decision. This is consistent with the object-oriented approach of program design and rapid prototyping recommended for example in C++ development, Stroustrup [70]. The ability to abstract up from a specific implementation to an implementation independent representation of the design of the program and then push down to multiple specific implementations is of considerable advantage for many programming tasks. Algorithm substitution for parallelisation and optimisation of programs for parallel platforms needs these two-abstraction level changing abilities.

Algorithm replacement for parallelisation based on program recognition has been addressed by Keßler et al. [2, 4, 5] Bhansali et. al. [6, 7, 8] and Di Martino et al. [1]. Keßler reports on implementations of these schemes in PARAMAT and SPARMAT. PARAMAT probably has the most extensive coverage in the number of non-trivial patterns "concepts" (100) using (160) templates Keßler [4]. SPARMAT Keßler [5] is a specialist sparse matrix enhancement that demonstrates that the technique can be applied to algorithms with a high degree of indirection and runtime data dependent data layout optimisation issues, where other more traditional prarallelisation approaches generally have poor results.

Di Martino et al. [1] reports on the differences in the approach adopted in the PAP tool against the work of Keßler. In their joint paper they conclude that the PAP recogniser is slower but more flexible and general than PARAMATS pattern recogniser.

1.3.3 Algorithm Learning Procedures

Wills [66] emphasises that the knowledge base of clichés was generated by hand in order to investigate the utility of the approach for a number of medium sized analyses on "student" programs. The automatic acquisition of the knowledge base appears to have been recognised as a significantly harder problem. There exists considerable work on generalised pattern matching, of which Nevill-Manning et al

[71] provides an interesting introduction, describing his application called sequitur. Naturally any cliché pattern located in a program by an approach such as sequitur would require additional meaning to be associated with it before it became a useful piece of program transformation knowledge. One approach to this might be to infer the function of such a program component through experimentation. A system would need the ability to fabricate test code to exercise the component and/or have the ability to deduce the functionality of the component by some abstract reasoning process.

One of the main restrictions to this possibility is the lack of a systematic method of building patterns. This omission from the works cited has been recognised by Villavicencio [72] and the beginnings of a method of achieving this are provided in his work.

1.4 Approach

The approach described in this thesis is based on observations of the processes that a human programmer performs when trying to discover the purpose, and errors in an unfamiliar source code.

For a procedural language such as FORTRAN, the programmer can take advantage of the probable partitioning of the program into subroutines that can encapsulate relatively simple algorithm components, although for some legacy code such structuring is not always available. For this work the examples used all have a procedural breakdown. While this breakdown is useful in the presentation of the examples, it is not a fundamental limitation of the approach.

The programmer will typically tackle a new source code problem using a variety of static analysis techniques along with specific dynamic test cases where the behaviour of code is examined during execution.

Examples of static analyses that a programmer may apply might be:

- Determination of the context in which each subroutine is called.
- Deduction of how each subroutine affects the data being manipulated.

- Identification of algorithms that are being embodied in the specific section of code.
- Construction of a call tree.
- Generation of outline flow charts.
- Formation of a view of the dependencies in the flow of control.

Dynamic analyses might include:

- Dissection of the code and exercise of the individual subroutines.
- Observation of how subroutines manipulate data at runtime using for example a debugger.

The programmer will have background knowledge of a range of algorithms in their domain of experience, but may be lacking specific experience of algorithms present in the source under analysis. Therefore reference to external information is a likely part of the overall process.

This thesis looks at two of these processes:

- Formation of a view of the dependencies in the flow of control.
- Identification of algorithms that are being embodied in the specific section of code.

The approach adopted to address the first of these processes is referred to as Reverse Traversal of the code. Specifically the analysis starts at the end of the program and works back through the flow of control tracing the significant data items thought he call tree. This is conceptually very simple and in practice the main difficulty appears to be the control and display of the dependency tree that results.

The approach to the second item is to implement a matching process whereby patterns within training samples of code are captured in a generic form and used as templates to match against unknown code samples. Matching against templates is probably only a small part of what a human programmer might do. Whilst significant

mathematical constructs might be familiar to a programmer and will stimulate hypothesis generation, the human programmer will also be able to “dry run” phrases of the code and match predicted behaviour against expectations within what might be termed the fluid hypothesis space.

Chapter 2 Learn Tool System Description

The "Learn Tool" is a set of analysis routines that seeks to assist the operator in evaluating the intent or purpose of legacy FORTRAN code. It was constructed during this work to allow experimentation with algorithms that could play a part in the machine understanding and transformation of source code. At this time the tool encompasses two approaches, Reverse Traversal and Pattern Matching.

Reverse Traversal is a bottom up approach intended to isolate the main calculation routes within a particular piece of source code. The main control flow of the source code is identified and then the last executed statement is used as a starting point. The analysis dry runs in reverse through the call tree, making notes on loop nests, scope and variable passing. As this traversal of the parse tree proceeds, sets of actions are triggered in response to particular types of statement.

Pattern Matching is a process of subjecting the parse tree to a direct comparison to a library of commonly occurring sequences that are associated with higher-level operations.

Both these processes have been built on top of the Sage++ Toolkit developed by "Indiana University, University of Oregon and University of Rennes" [49][50]. This library includes a FORTRAN parser capable of accepting a variety of FORTRAN source forms including FORTRAN77 and FORTRAN90, and a C++ class library for interrogating and manipulating the parse tree.

The following sections will deal with the Sage++ system, Reverse Traversal and Pattern Matching in more detail, and conclude with an overview of how the Learn Tool is operated.

2.1 Learn Tool History and Composition

The Learn Tool system, created for this work, started as two separate analysis programs written C++ on Digital UNIX and Solaris using Motif™. These have been ported to a single MFC application, LearnToolViewer, in the later stages of the work and it is in this form that example output will be presented in this document. The source code is approximately 10,000 lines of C++ including comments but excluding the Sage++ library that required a few modifications to port to a windows platform. The viewer with its graphical outputs and dialogs represent about a fifth of the whole.

2.2 The Sage++ System

This section provides a brief overview of the Sage++ system. This is intended to provide a convenient reference to some of the terminology used in the following sections that describe the algorithms and implementation details of the Learn Tool.

The Sage++ system adopts a project concept that points to one or more dependency files generated by the FORTRAN parser program *f2dep*. The dependency files contain the information about the parse tree of an individual file within a project. At the top level the project consists of a text file that lists the component dependency file paths. The Sage++ API provides a method of instantiating an SgProject object with reference to the contents of the project text file. All the subsequent objects can be accessed from the SgProject object. The SgProject object contains one or more SgFile objects that in turn contain the SgStatement objects (one for each line in the source code). SgStatement(s) can be flow control items representing for example the beginning and end of DO loops, program statements, declarations, assignments and subroutine calls etc. SgStatement(s) refer to their component SgExpression(s), SgSymbol(s) and SgType(s).

The SgStatement object has lexnext() and lexprev() functions that allow the application to traverse forwards and backwards through the statements within the source code. The SgFile object allows random access to any of the SgStatement(s) it owns.

Each different type of SgExpression, SgStatement is implemented as an object specialisation with its own unique access functions and dynamic typing is supported within the Sage++ API through global functions. So for example the application can determine if a particular statement is a program header by calling the function *isSgProgHedrStmt* on the SgStatement object in question, this returns a pointer to the object if it is and NULL if it isn't.

2.3 Reverse Traversal

This section describes how the reverse traversal is performed and presents and explains the output achieved.

The reverse traverse analysis searches the parse tree and locates the statement that is the last to be executed. This statement is used as a starting point for the analysis, which then works back statement-by-statement following the logical flow of control of the program. The traverse is terminated when the first statement is reached.

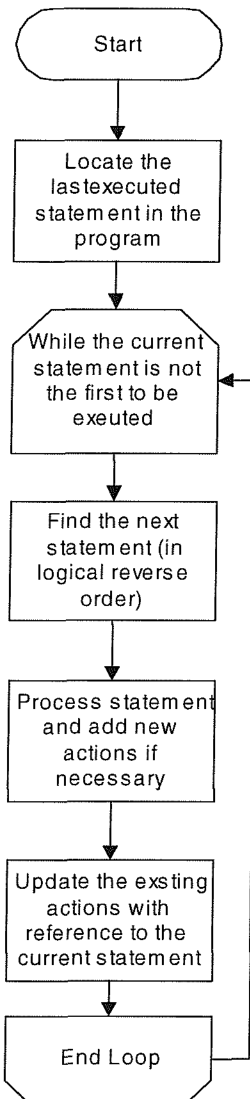


Figure 3 Reverse Traverse Flow Chart

The model adopted for the reverse traverse process is to imagine a programmer examining the source code and making notes on the flow of control, the significance of specific variables and identification of the core calculations in the main algorithms.

For each statement visited an exprSummary object is created for the statement and the expressions within the statement. The exprSummary simplifies the task of discovering if an item being looked for actually exists within the statement. The exprSummary provides a filter for the information in an SgStatement so that actions can be targeted at specific parts of the statements.

As the analysis progresses through the statements, actions are added to an actionList in response to certain conditions. In our model the programmer has noted for example that a particular variable or data structure should be traced back to its declaration and any manipulations on it should be recorded. Any existing live actions in the actionList are updated if a new piece of information is available from the current statement that is relevant to that particular search. A flow chart of the main analysis loop is presented in Figure 3.

The design of the algorithm is based around the ability to extend the number and type of actions available although it was only taken as far as the reverse traverse process

of identifying the program outputs and tracing back to the declarations and intervening manipulations before the focus of the work shifted to pattern matching.

The reverse traversal follows the logical flow control of the program (in reverse) and as it proceeds it maintains a context structure indicating the control level of each statement within the program as a whole. This is effectively a call stack and is essential for being able to match data on multiple levels that may have their variables names changing or going out of scope in subroutine calls.

This general architecture has been used to extract information from the program, particularly concerning how the results of the program have been arrived at. A new action is created for each output statement, for example writing results to a file or the screen. The action is to keep track of the data being output and if it occurs in any statement earlier in the code to keep a note of that occurrence and any inter-dependence with data associated with that statement. This is achieved by maintaining a dataTrace with each action that records the statements that reference the variable being traced, or any variable that is used to influence the result. The dataTrace consists of an ordered list of dataTraceItem(s) that grows, each time a statement is visited that has a relevant reference to the items being traced. The traces operate over the whole code by following subroutine and function call parameters and mapping them to the actual variables at the calling level.

Scalar and array items are tracked and some filtering is provided to discount rarely executed paths in the code, for example conditionally executed error reporting. Whilst this may omit important information in some cases, the resulting dataTrace can remain linear.

During the reverse traversal, all items that have already been identified as contributing to the output item are search for, which could lead to an exponential growth of the tree. By ensuring that if a complete assignment has been made then the assignee is removed from the scope of the search moderates this potential for algorithm runaway. For example in a CALL, SUBROUTINE pair of statements, the parameters in the CALLs parameter list go out of scope (for the search) and the corresponding parameters in the subroutine parameter list come into scope.

Information about the type of statement that they relate to is stored in the dataTrace items and each item has a reference to the "parent" searched for item. This information can be used to construct a graph of the overall dependence flow though simple programs. This output form has been chosen as it illustrates how rapidly the complexity of the analysis grows for relatively simple algorithms. This dependency graph is not a true dependency analysis of the sort described by Wolf [21][22] but more an Inter Procedural or data flow analysis in the style of Merlin [48] and Walker [35].

The Reverse Traversal process is intended to mimic the code inspection process that humans employ to understand the purpose and flow within a code. It achieves a transformation of the code representation into a graph that highlights the statements that have significant influence on the results of executing the code. It provides a view of the code as a whole unencumbered by a particular procedural decomposition. Its use within the program recognition process was initially intended to flatten the program structure and pre-filter the parse tree before the pattern matching process. This has been achieved, however the increase in complexity of representation for relatively simple programs caused an adjustment to the planned approach adopted for the pattern matching process removing this flattening step.

2.4 Pattern Matching

The pattern matching technique is described in this section. A detailed explanation of the structures and matching algorithms developed follows an overview of the whole approach.

2.4.1 Related Work on Pattern Matching

There exists a wide body of work relating to pattern matching covering subject areas diverse as genetic sequence matching, web searching to computer science re-writing systems. The class of pattern matching used in this work seems to be closest to the work described by Kucherov [73]. Kucherov describes the use of a Directed Acyclic Word Graph (DAWG) however the algorithm is restricted to finding the first match to a particular pattern. Closely related, from a computer science perspective is the large system simplification work of Baker et al. [74],[75] and [76] that applies pattern matching to highlight possible redundant sections of code that could be

unified for example in a common procedure call. The approach adopted in this work is a rudimentary technique aimed at limiting the exponential growth of the search space by discarding potential sequences as soon as they fail the overall matching criteria.

The description by Hagemeister et al. [8] comes closest to the approach used in this work. Hagemeister develops a syntax for describing the patterns, which is based very closely on the tokens available from the Sage++ parser. This approach in turn is based on the SCRUPLE matching described by Paul et al. [77]. The pattern matching approaches in PARAMAT [4] appears to be based on the graph parsing approach of Wills [66] whereas the PAP work of Di Martino et al. [1] is closer to the SCRUPLE technique and the method described herein.

More recently direct AI approaches have been attempted for pattern matching. Quilici A., Yang Q., et al [78] report on their work and conclude that direct application is not the best approach, however they can be made to be effective if constraint satisfaction techniques are included.

2.4.2 Overview of Approach

The pattern recognition process involves breaking down the code into a sequence of tokens that contain sufficient information to retain the algorithmic content of the original source code and yet is simple enough to be used in a pattern-matching algorithm. The Sage++ toolkit presents the information about the code as a hierarchy of objects. To unwrap this hierarchical representation, a recursive algorithm has been created that generates a flat list of items from identified sections of the source code.

This flat structure is incorporated into a CodeStatus object, which contains a list of codeStatusItem(s) that are used to match against. Individual matches occur on integer values and no string matching is needed making the process fairly efficient.

The flat list of items is generated by stepping through the statements in the order that they appear, in the code. Each statement generates a list of items, which is appended to the current list. Within a statement, depending on the type of statement, expressions, symbols and types may also generate lists of items, which are inserted

into the full list produced by the statement. Overloading the constructor on the CodeStatusItem class facilitates the process of recursive generation.

The selection of which items are placed into the CodeStatus object has evolved as the range new concepts to recognise has grown.

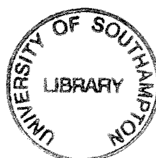
The pattern library is managed under a knowledgeBase class; each pattern in the knowledge base is referred to as a Rule item. Each Rule has a series of Keys that are used to match to specific code examples. The Keys are built from and subsequently matched to the CodeStatusItems(s) in the flat format expansion of the parse tree. The Keys are related to each other in a Rule with a simple syntax.

The Rule syntax supports the following features:

- CodeStatusItem type and order of occurrence
- Relative positioning of items is either strong or weak
- Instance correlation is enforced
- Multiple instances of the same rule match with the same starting item are flagged as voiding the match.

The user constructs rules by picking CodeStatusItems derived from a test sample of code. Once a CodeStatusItem is incorporated into a Rule it becomes a component Key. The Keys are selected in a particular order and the following operators ("AND", "THEN" and "LAST") are used to indicate the relative positioning relationship. The "AND" operator indicates the spacing between the currently selected item and the next one must be exact for a match to be possible (strong relative positioning). The "THEN" operator indicates that once this item is matched then the next item to match can be an arbitrary distance from the current item (weak relative positioning). The "LAST" operator terminates the sequence.

Each Rule is currently allowed to be referenced to one of its Key items, usually a variable symbol, that has to have a common matching element during subsequent pattern matching operations. For example the referred to item in a "counter" Rule is the counter variable.


UNIVERSITY OF SOUTHAMPTON LIBRARY

The "OR" operator although not explicitly supported has been implemented in the Learn Tool pattern matching algorithm by allowing several rules to be equivalent as far as the matching process is concerned. This is managed by a Rule association process, and simply allows Rules to match with items of either Rule A or Rule B if Rule A and Rule B are associated. This has been particularly useful when matching expressions that contain commutative operations. It can also be applied to assist the pattern matching as an alternative to code normalisation. This is similar to PARAMAT, Keßler [4] in that several routes exist to the final higher level pattern match.

2.4.3 Code Status Items

For a section of code a flat list of CodeStatusItem(s) is generated by moving from statement to statement and conditionally adding items to the list depending on the type and content of the statements. CodeStatusItems(s) record five fields for each record. These are described in table Table 1.

Item	**Description**
Type	The type field records whether the item is derived from a 'Statement', 'Expression', 'Symbol' or 'Rule'. The first three are generated from the basic parse tree supplied by the Sage++ project. The 'Rule' items are added when the matching process has located a match to a 'Rule' from the knowledge base.
Variant	This field records what type of 'Statement', 'Expression', 'Rule' etc. this instance is. e.g. a 'Statement' might be a PROC_HEDR.
Id	This field records the specific instance of the occurrence of this item.
Tag	The tag is a string containing a textual representation of the variant.
Name	The name is a string that is used if the specific instance has a token associated with it. For example a 'Symbol' type, variant VARIABLE_NAME would have a tag of "VARIABLE_NAME" and a name indicating the name of the variable in the code. In the case of a symbol derived item the name will be directly associated with the Id of the item.

Table 1 Description of the fields in a "CodeStatusItem".

The codeStatusItems are used for both creating trial Rules as well as matching against Rules that have been previously formulated. The Tag and Name items are included to allow intelligible feedback to the operator on where patterns have matched and why. The comparison operations are restricted to the Type, Variant and Id integer values that uniquely identify the item in the sequence. The use of three integers for the comparisons allows most compare operations to complete (with a negative outcome) after a single integer compare instruction. The attention to this detail is mainly relevant because the speed of operation of the matching algorithm is quite important for allowing such as system to be realised as a practical product.

2.4.4 Rules

Rules contain the matching information that has been formulated from example sequences of code. The matching sequence is stored as a list of Keys that are used to mach against codeStatusItems. The rules maintain information about other rules that have been marked as being equivalent, enabling the formulations to include an OR concept. Each Rule has a specific instance ID from one of its Keys that characterises the Rules ID during the matching process.

2.4.5 Keys

Keys maintain information about the type and instance from the originating codeStatusItem. They also maintain information about their relationship to the next Key in the sequence within the Rule. This takes the form of the "THEN", "AND" and "LAST" operators that are owned by the current key and relates to the next Key in the sequence. In the case of the "AND" relationship the separation of the Keys is stored. A Key can be considered to be a single character in a string matching problem, with the "AND" spacing being a fixed distance and the "THEN" being the variable length of don't cares of Kucherov [73].

2.4.6 Matching Algorithm

The matching algorithm allows the multiple compares to be processed in a relatively efficient and memory conservative way. The overall process is outlined in Figure 4. For each rule currently in the database the current codeStatus structure is examined for instances of each of the keys that make up the rule. For simple rules this might be

a VARIABLE_NAME or an assignment operator for example. For more complex rules the key instances may be previously matched rule items. A table is constructed of all the possible instances of the searched for keys. This is then traversed key by key building valid sequences that match the partial rule sequence up to the point of the current key. Invalid sequences are removed, as the first mismatch is located. Once all the keys have been evaluated a final list of sequences remain that are matches to the current rule.

At this point the uniqueness of the sequence matches is checked. If the same starting key is present for multiple matches then the Rule as currently defined is ambiguous for the code being evaluated and requires re-formulating.

Only unique instances of rule matches are currently reported in the Learn Tool program. This uniqueness problem is addressed by Paul et al [77], by delivering the "shortest match" permutation from the matching engine. They comment that patterns that have a combinatorial explosion problem are rarely found in program understanding problems, however that is probably critically dependent on the skill of the operator forming the rules.

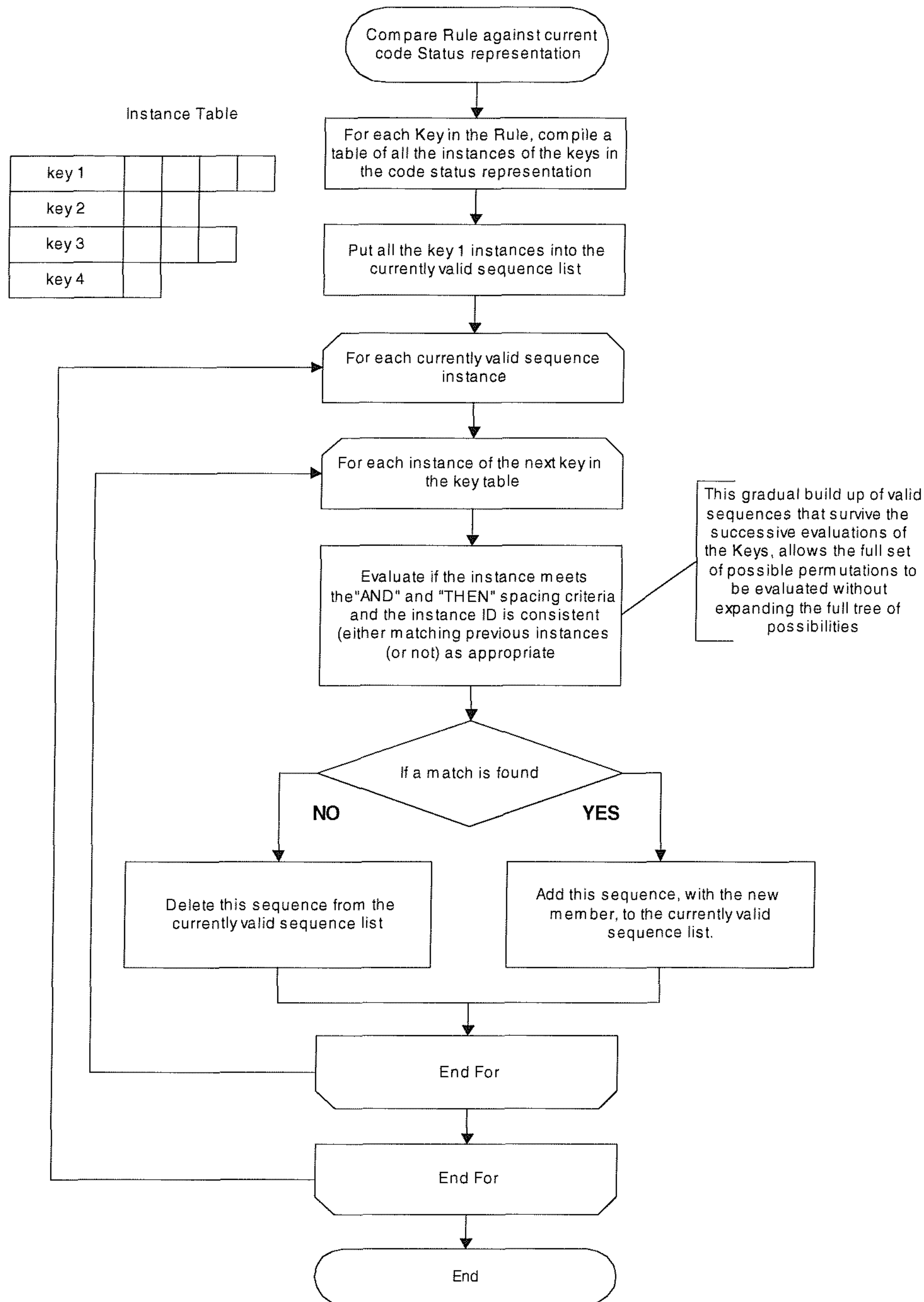


Figure 4 Flow chart detailing the rule-matching algorithm

2.4.7 Learn Tool Facilities

The Rules (patterns) are stored in a list belonging to the knowledgeBase class. This provides the Load/Save functionality and access to the patterns during a matching operation. Users may build test patterns interactively and add them to the knowledgeBase class using a Rule Editor Dialog.

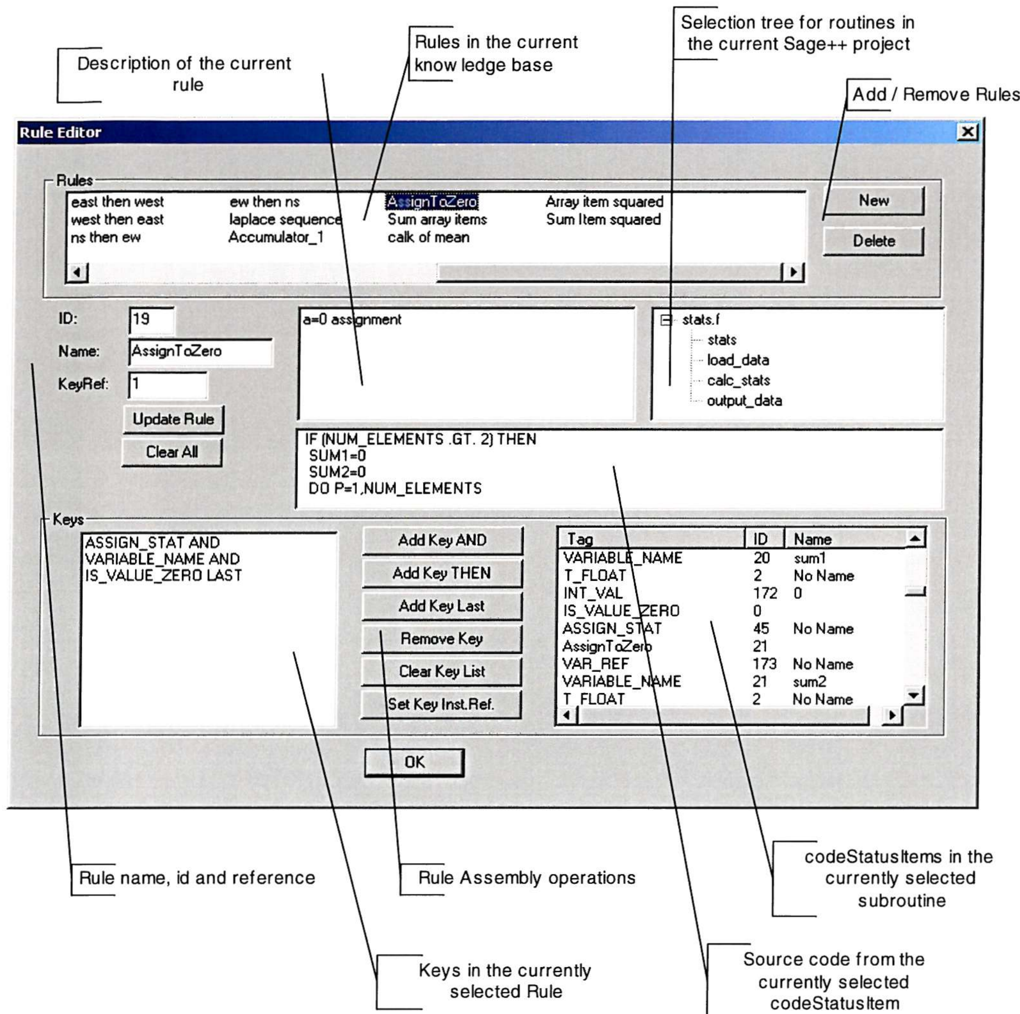


Figure 5 Annotated Rule Editor Dialog

Figure 5 shows the Rule Editor dialog in the LearnToolViewer application. It is showing the construction of the AssignToZero Rule within the context of the Statistics example, which is addressed in more detail in the next chapter. Currently all rules exercised in this work have been constructed by hand and have been used to perform what if experiments on combinations of keys that work and those that don't. A systematic approach and guidance on how to construct rules would be essential for a fully developed system. It might be possible to import rules from other pattern matching work, for example the pattern library of Kessler [1] or Di Martino [2], although as this approach excludes code normalisation it is doubtful whether the libraries could be transferred without significant modifications.

Associations between Rules are established using the Rule Association Dialog. An example of this is shown in Figure 6, for the "North Then South" Rule that will be discussed in more detail in the "Laplace" example in the next chapter.

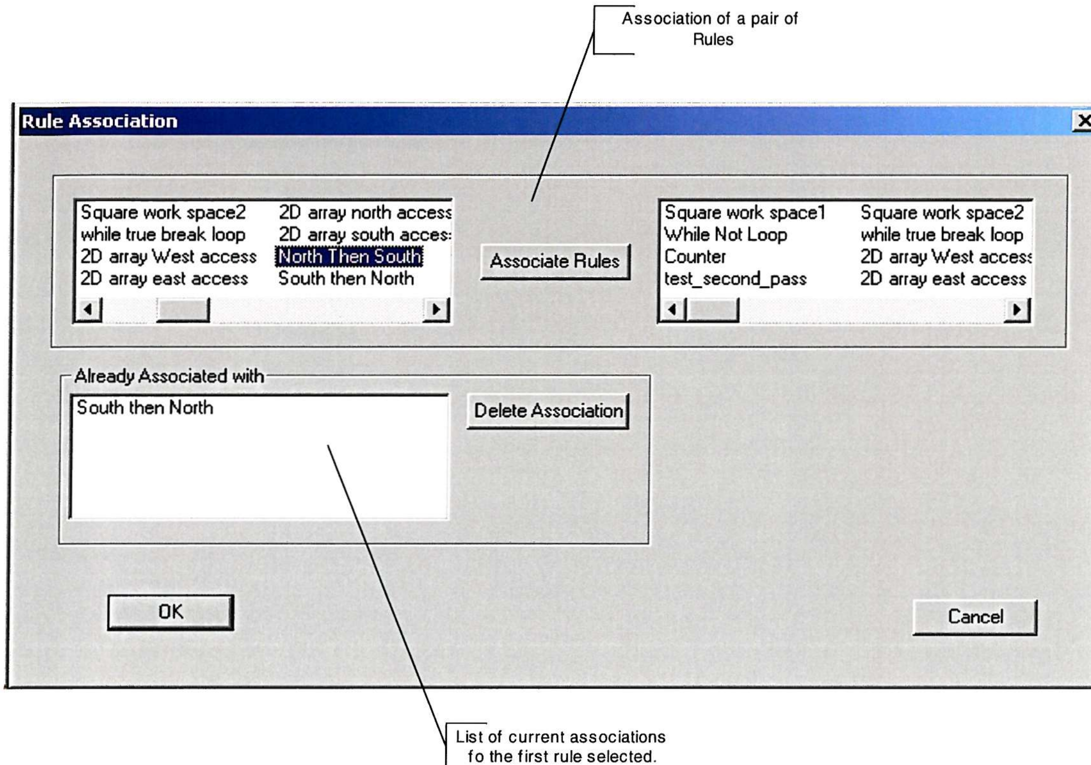


Figure 6 Annotated Rule Association Dialog

For both the Reverse Traverse and Pattern Matching the Learn Tool provides simple graphical output showing the dataTrace data and the pattern matching detail respectively. In a completed parallelisation Toolset this information would be available to the user in a number of forms and would stimulate the tool to provide options for algorithm substitution with perhaps an indication of the potential benefits that might be realised for any particular substitution.

Chapter 3 Test Cases

The test cases presented here start with the source of the program used to generate the Rules. The reverse traverse of the code is shown followed by the matching results against the Rules and a description of the component rules. This is followed by example variations of the source example to demonstrate the properties of the rules being investigated.

3.1 Statistics Example

This first example is of a simple statistics program that generates the mean and variance of a set of data. The data is read from a text file and the results are computed and written to another output text file. The source code is given in Appendix 1a.

3.1.1 Reverse Traverse

The reverse traversal of this example provides two dependency graphs, one for each of the mean and variance.

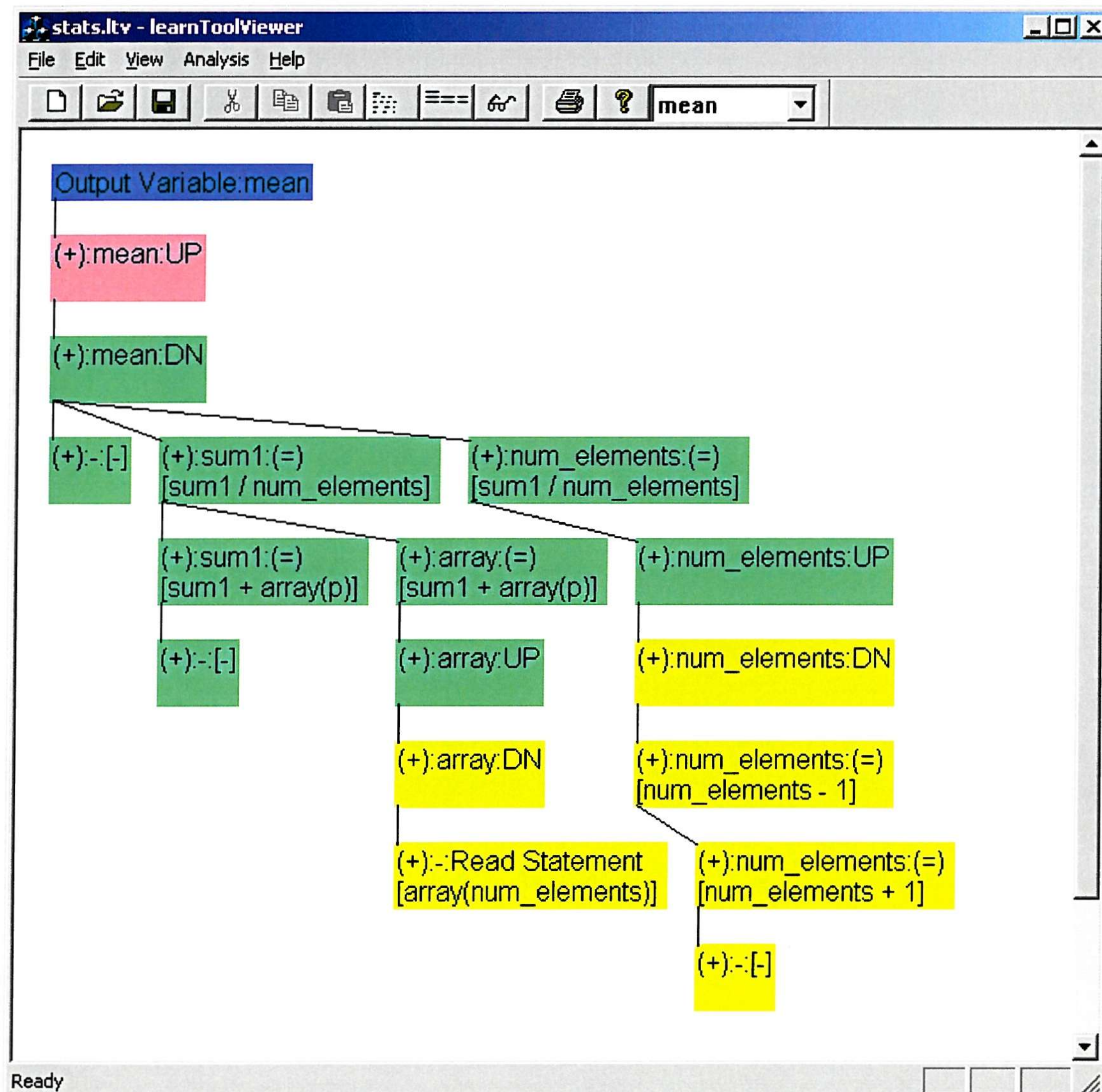


Figure 7 Screen shot of Stats example reverse traverse showing mean dependency graph.

Figure 7 shows the main screen of the learnToolViewer application after it has performed a reverse traverse analysis of the example code. At the top of the diagram the output variable "mean" is the root (from line 130). This variable is traced though an upward CALL/SUBROUTINE paring into the main program and then down into the subroutine that performs the calculation. At this point the mean is found to be dependent on an initialisation denoted by the (+):-:[-] string, the variable sum1 in the expression (sum1/num_elements) and num_elements in the same expression. These variables in turn are followed and their dependence on the input variable "array" is located after another change of subroutine scope.

The change of colour in the display is used to show the call stack level of the code. In this example the top of the tree is in purple and occurs in the "OUTPUT_DATA" subroutine off the main program level. The main program level is shown in red, the "CALC_STATS" subroutine level is in green and the "LOAD_DATA" level is in yellow.

The output syntax of the string for each node starts with a "(+)" if it is a new node in the tree. The next item, spaced with a ":" is the item reference, for example "mean" or if it is an initialisation a "-" is used. The next item again separated by a ":" is an "(=)" if there is an assign association, an "UP" or "DN" for a subroutine call association. If there is an "(=)" association then a string is appended with the assignee enclosed in square brackets. For an initialisation a blank "[-]" is appended.

Figure 8 Reverse Traverse dependence graph for variance.

Figure 8 shows the similar dependence graph for the variance output variable. Whilst the calculation of the variance is mathematically fairly simple, the representation of the dependence graph in this form has become quite extensive.

3.1.2 Pattern Matching

The pattern matching approach in this example has concentrated on the calculation of the mean. The rule for this is detailed in Figure 9.

```
Rule 21 : Name: calculation of mean
Keys:{
      ARRAY_REF AND
      VARIABLE_NAME AND
      STAR_RANGE AND
      VARIABLE_NAME THEN
      Sum array items THEN
      ASSIGN_STAT AND
      VARIABLE_NAME AND
      DIV_OP AND
      VARIABLE_NAME AND
      VARIABLE_NAME LAST }
```

Figure 9 Rule: calculation of mean

The first two items establish the id associated with the array that stores the items on which the mean is to be calculated. The STAR_RANGE is specific to the FORTRAN calling scheme whereby the array is a reference to data passed into a subroutine that may change depending on from where it is called. The second VARIABLE_NAME item is the range index and the divided by element in the mean calculation. The "sum array items" is a rule that is expanded below. The last five items establish the final division and assignment.

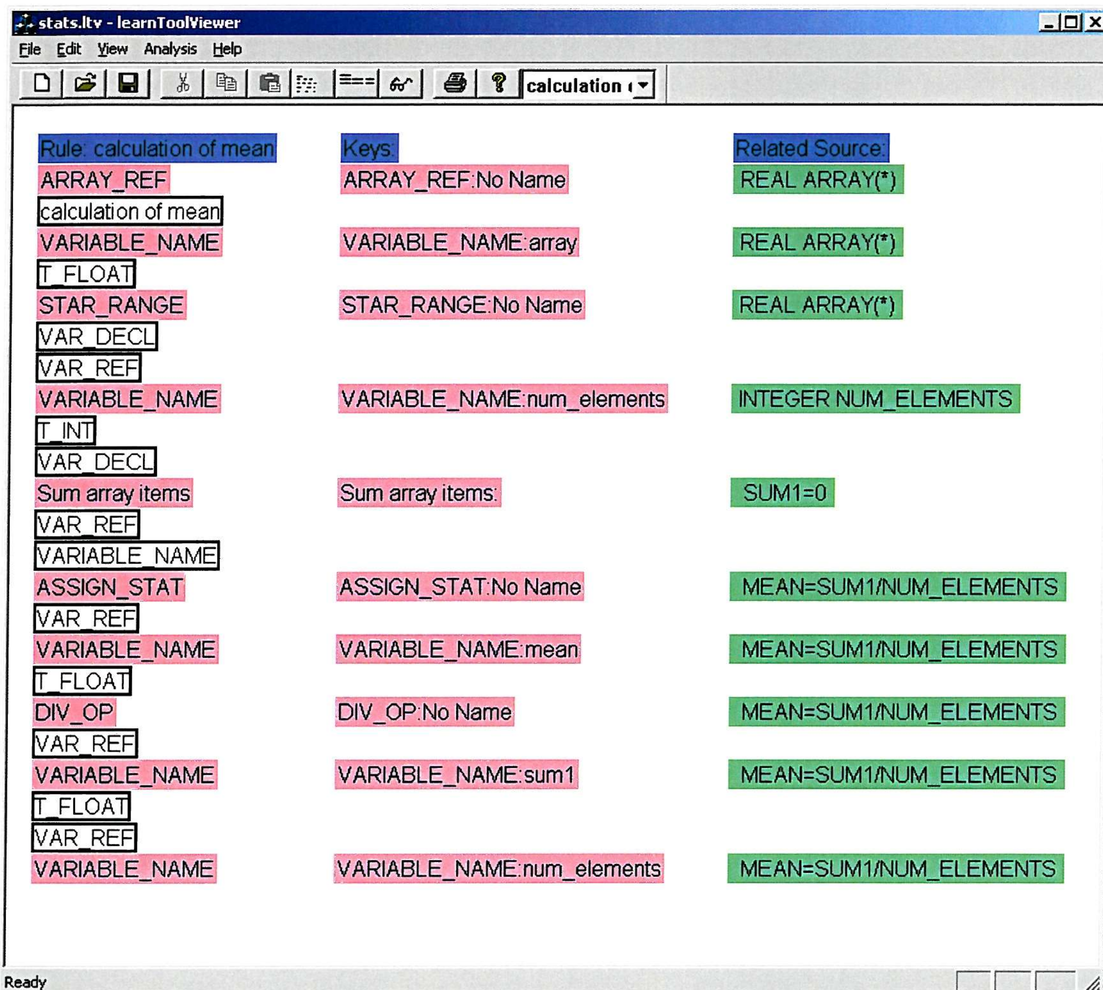


Figure 10 Screen shot of the match detail for the calculation of mean rule.

Figure 10 shows the output from the pattern matching within the learn tool for the calculation of mean rule.

```
Rule 20 : Name: Sum array items
Keys:
{       AssignToZero THEN
        FOR_NODE AND
        VARIABLE_NAME THEN
        Accumilator1 AND
        ARRAY_REF AND
        VARIABLE_NAME AND
        VARIABLE_NAME THEN
        FOR_NODE LAST}
```

Figure 11 Rule: sum array items

Figure 11 shows the keys for the "sum array items" rule. The assignToZero is a rule that captures the initialization of the variable to be used as the storage for the sum. The FOR_NODE establishes the loop within which the sum is performed. The VARIABLE_NAME refers to the loop index and the Accumilator1 is a rule that captures the "a=a+b" construct. This is augmented with a reference to the data VARIABLE_NAME item and the loop index variable name. The closure of the loop provides the last key for the rule. Figure 12 shows the screen shot for this rule.

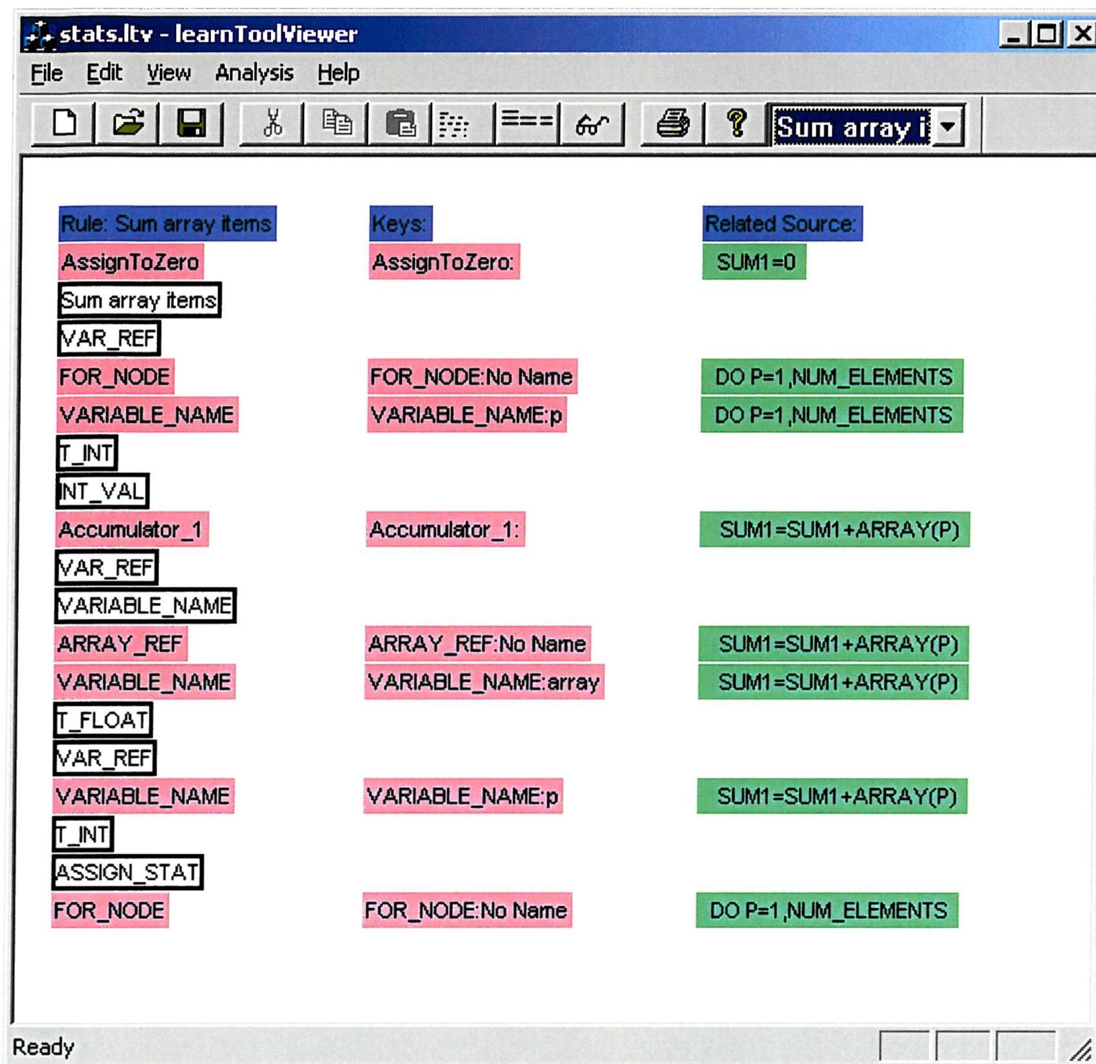


Figure 12 Screen shot of the match detail for the Sum array items rule.

```
Rule 19 : Name: AssignToZero
Keys:
{       ASSIGN_STAT AND
        VARIABLE_NAME AND
        IS_VALUE_ZERO LAST }
```

Figure 13 Rule: Assign ToZero

The keys of the AssignToZero rule are shown in Figure 13 and the corresponding screen shot is included in Figure 14.

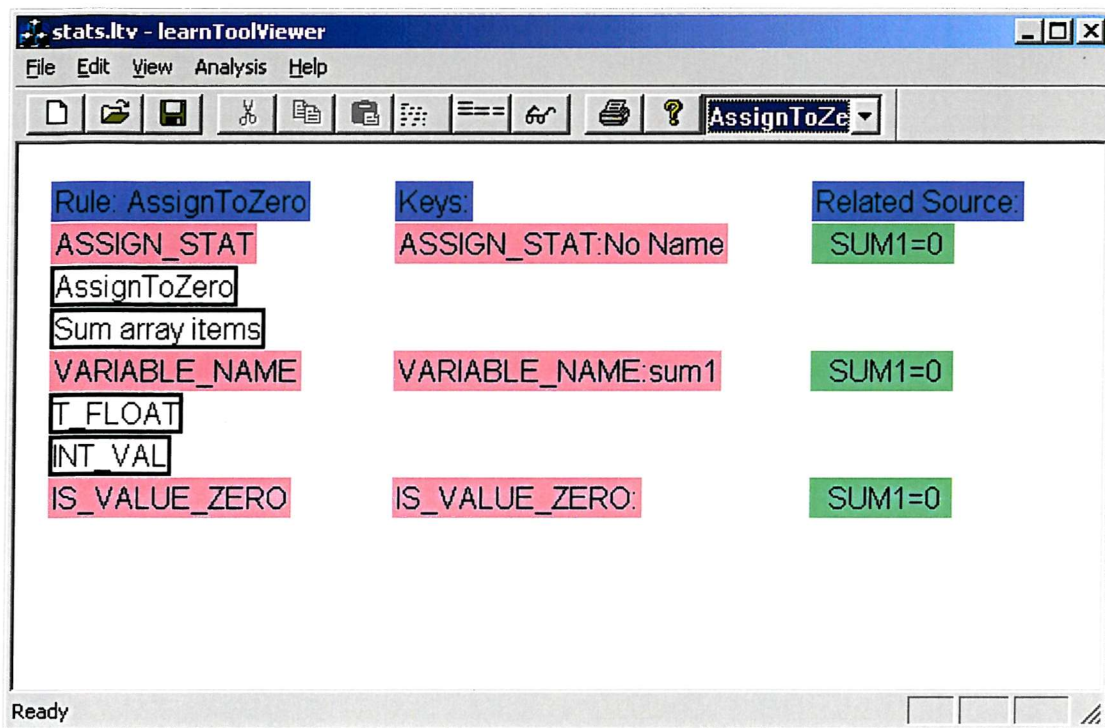


Figure 14 Screen shot of the match detail for the AssignToZero rule

```
Rule 18 : Name: Accumulator_1
Keys:{
      ASSIGN_STAT AND
      VARIABLE_NAME AND
      ADD_OP AND
      VARIABLE_NAME LAST }
```

Figure 15 Rule: Accumulator_1

Figure 15 and Figure 16 complete the set of rules used to match against the calculation of mean.

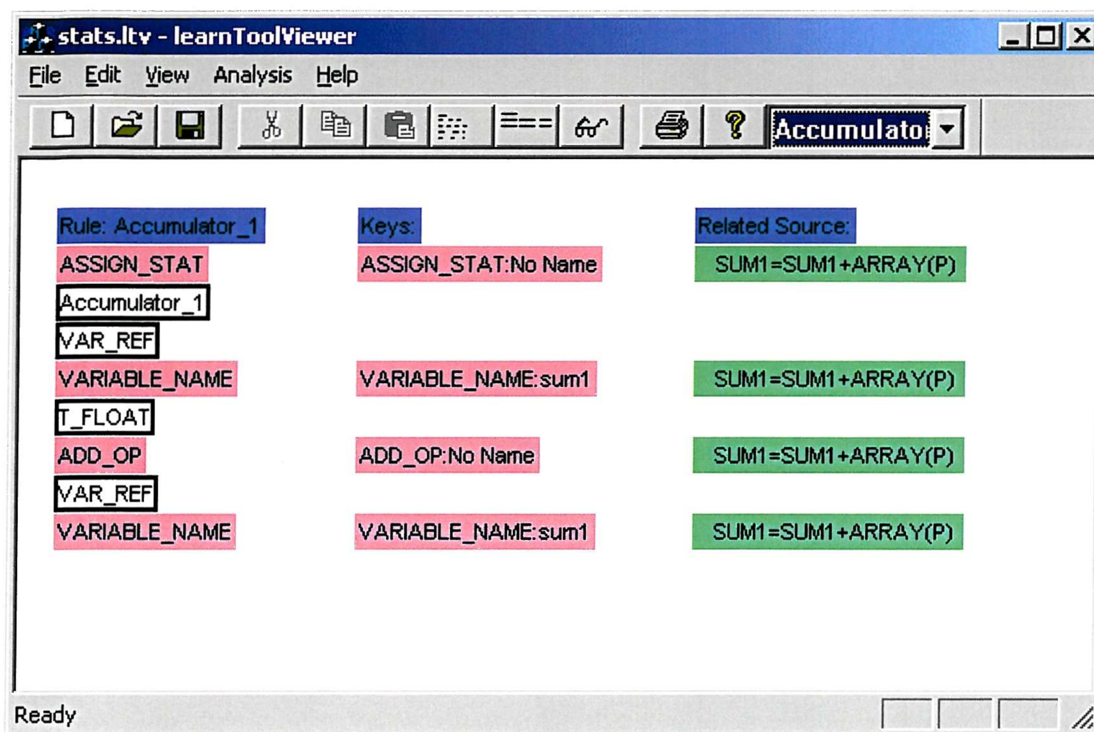


Figure 16 Screen shot of the match detail for the Accumulator_1 rule

3.2 Laplace Example

This example examines a simple two-dimensional relaxation algorithm. A regular grid is used with its edges set to a pre-determined boundary condition. A numerical iteration process is used to calculate the internal points that satisfy Laplace's equation:

$$\frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y} = 0$$

The source code is given in Appendix 1b. Almost all relaxation algorithms operate by implementing an averaging scheme so that the values in each element are updated on the basis of the values in their neighbours. This update occurs in an iterative loop until the evaluation of the maximum error falls below a pre-determined threshold. The averaging construct in this case is embodied in lines 183 and 184 of the source. A more detailed discussion of relaxation algorithms can be found in Teukolsky et al [79] chapter 19.

3.2.1 Reverse Traverse

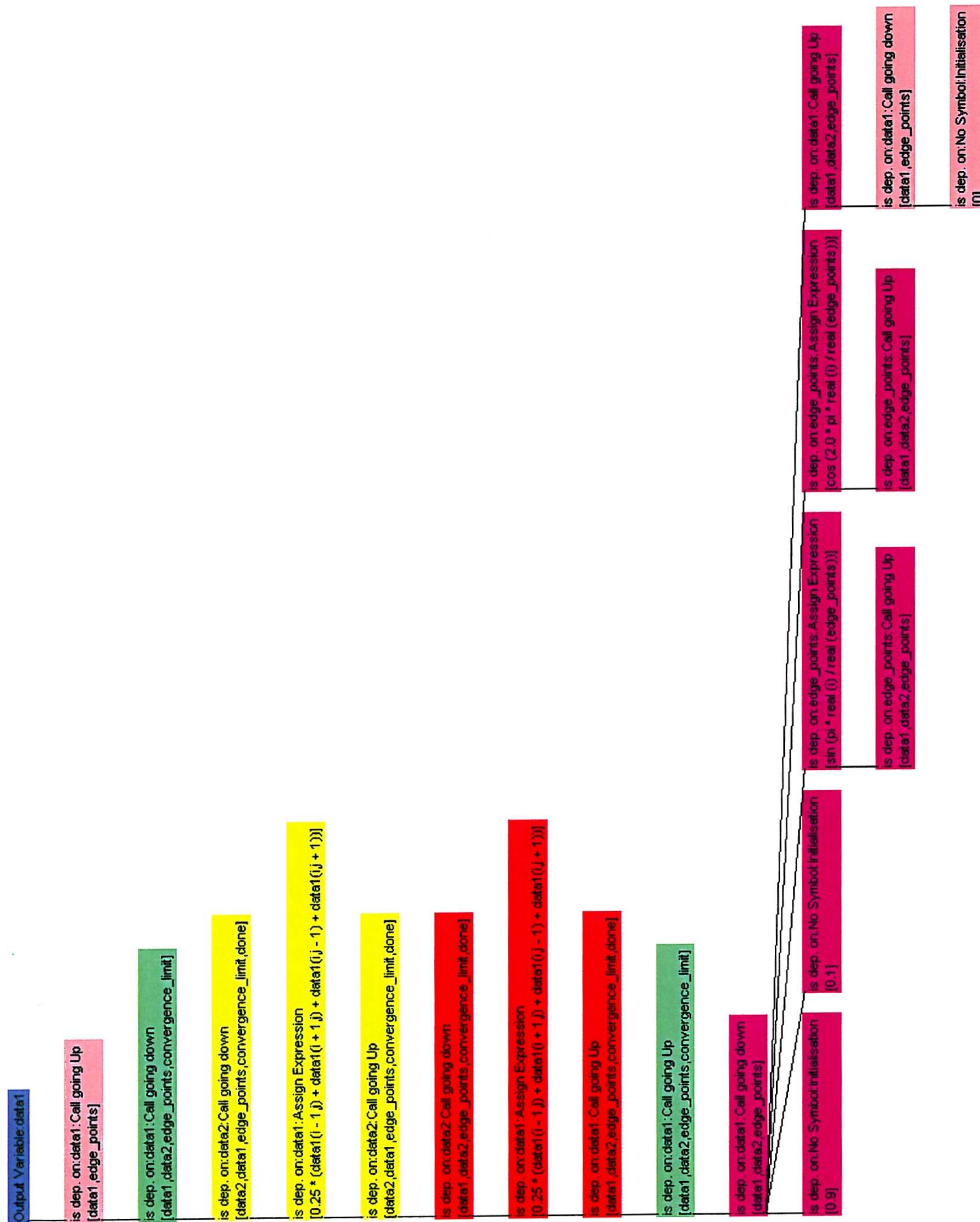


Figure 17 Reverse Traverse dependence graph for data1

Figure 17 shows the reverse traverse dependence graph for this example. There is only a single identified output "data1" and this is traced through the two calls to the itterate_step subroutine and the initialisation routines where the boundary conditions are set-up. The lack of a trace item for the initialisation of the edge_points variable is due to it being set in a PARAMETER statement for which there is no method for

generating an expression summary. This appears to be a limitation of the Sage++ parse tree.

3.2.2 Pattern Matching

In this example the core pattern to match against is the averaging statement embodied in lines 183 and 184. The rule for this is shown in Figure 18.

```
Rule 17 : Name: laplace sequence
Keys: {
      FOR_NODE THEN
      FOR_NODE THEN
      ew then ns THEN
      FOR_NODE THEN
      FOR_NODE LAST }
```

Figure 18 Rule: laplace sequence

```
Rule 16 : Name: ew then ns
Keys: {
      west then east THEN
      North Then South LAST }
```

Figure 19 Rule: ew then ns

The repeated FOR_NODE items provide the context for the main key, which identifies the East West then North South array access pattern.

```
Rule 14 : Name: west then east
Keys: {
      2D array West access THEN
      2D array east access LAST }
```

Figure 20 Rule: west then east

```
Rule 7 : Name: 2D array West access
Keys: {
      ARRAY_REF AND
      VARIABLE_NAME AND
      SUBT_OP AND
      T_INT AND
      IS_VALUE_ONE AND
      T_INT LAST }
```

Figure 21 Rule: 2D array West access

The "ew then ns" rule is constructed on top of two sub rules, the "west then east" (Figure 20) and the "North then South". Similarly the "west then east" rule is built from the "2D array West access" (Figure 21) and the "2D array East access". Each of these rules has a reversed counterpart, for example "North then South" has a partner "South then North" which are labelled as equivalent for the matching process. The equivalence of these rules allows the matching to work independently of the order of the coding of the array accesses.

Figure 22 Shows the output from the rule matching process for the "laplace sequence" rule. The contributing rule 2D array West access is shown in Figure 23.

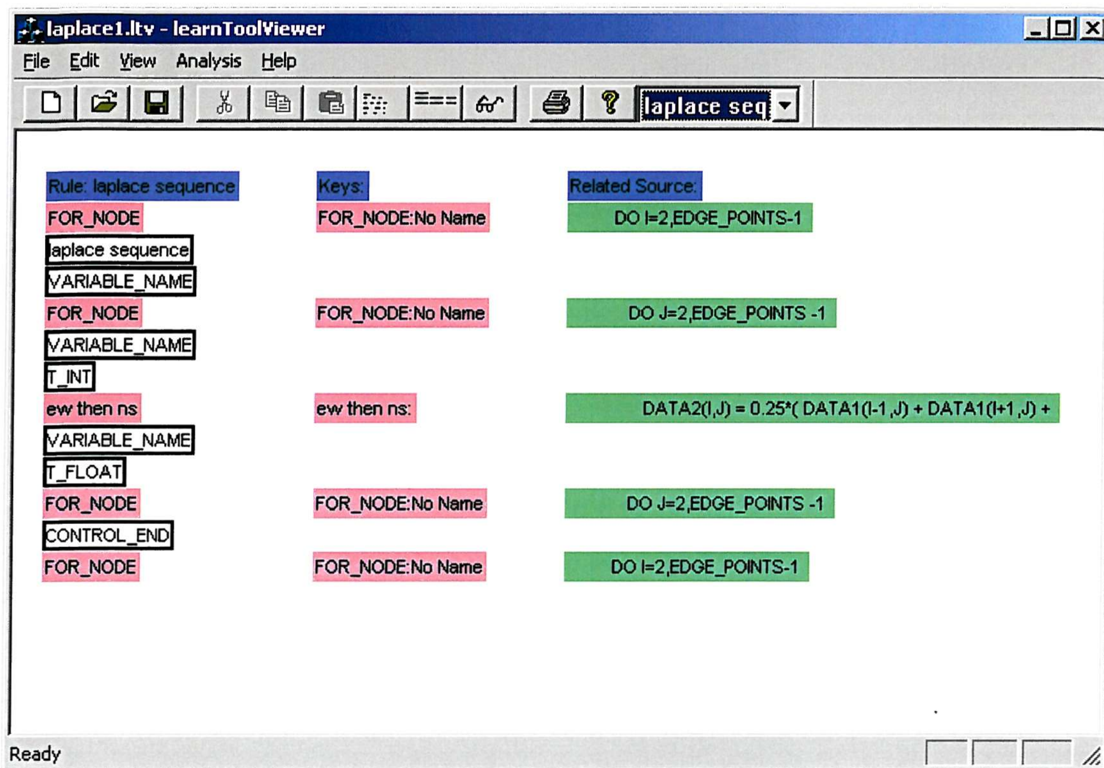


Figure 22 Screen shot of the match detail for the "laplace sequence" rule

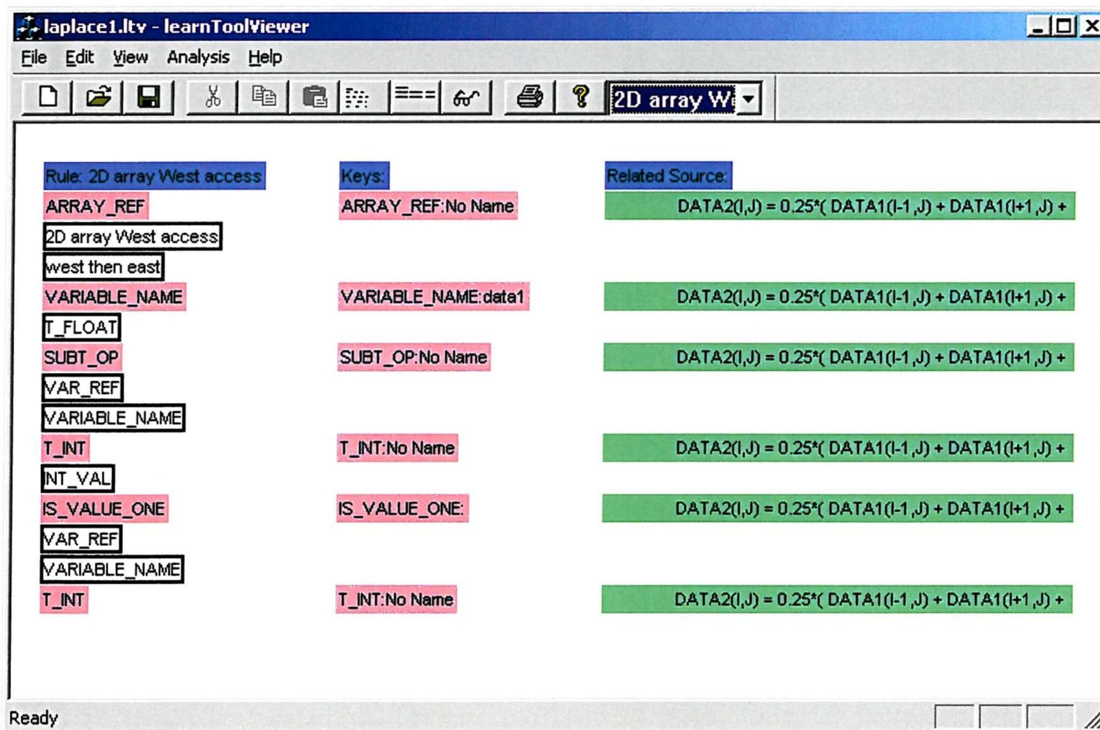


Figure 23 Screen shot of the match detail for the 2D array West access rule.

3.2.3 Laplace Variation 1

The first variation to test is to split the averaging process using a temporary stack variable to accumulate the sum. This is achieved by replacing lines 183 and 184 with the code in Figure 24.

```
ACCUMULATOR  = DATA1(I-1,J)
ACCUMULATOR  = ACCUMULATOR  + DATA1(I+1,J)
ACCUMULATOR  = ACCUMULATOR  + DATA1(I,J-1)
ACCUMULATOR  = ACCUMULATOR  + DATA1(I,J+1)

DATA2(I,J) = 0.25 * ACCUMULATOR
```

Figure 24 Variation 1 coding of Laplace example

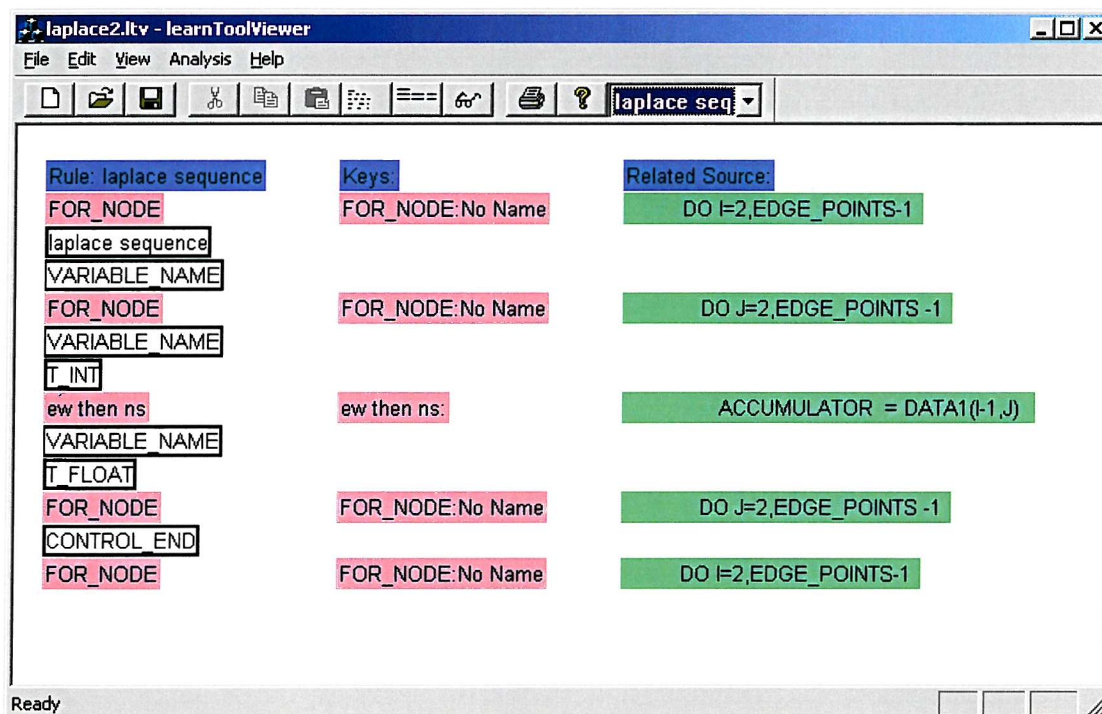


Figure 25 Screen shot of the match detail for the" laplace sequence" rule for this variation

Rule matches the "laplace sequence" rule correctly for this variation of the code as is shown in Figure 25.

3.2.4 Laplace Variation 2

```
             DATA2(I,J) = 0.25*( DATA1(I-1,J) + DATA2(I+1,J) +
    +                             DATA1(I,J-1) + DATA2(I,J+1) )
```

Figure 26 Variation 2 coding of Laplace example

Figure 26 shows the second variation coding of lines 183 and 184 of the original example. This time the matching algorithm correctly does not match to the "laplace sequence" rule. It recognises the individual North, South, East and West 2D access rules but the combinations of these for example "west then east" rule is not matched because the expected common variable name "DATA1" is not common in this instance of the sequence.

3.2.5 Laplace Variation 3

```
             DATA2(I,J) = 0.25*( DATA1(I-1,J) + DATA1(I+1,J) +
    +                             DATA2(I,J-1) + DATA2(I,J+1) )
```

Figure 27 Variation 3 coding of Laplace example

Figure 27 shows the third variation coding of lines 183 and 184 of the original example. This time the matching algorithm correctly does not match to the "laplace sequence" rule. It recognises the two pairs of "North then South" and "East then West" but cannot link them together because of the change of variable name between the instances.

3.2.6 Laplace Variation 4

Figure 28 shows the last variant of the Laplace example. In this case all the elements for the "laplace sequence" rule are present and it is only the minus sign between the B and the C on the last line that prevents this variation from being a correct coding of this part of the algorithm. The Learn Tool achieves a match against the "laplace sequence" rule as shown in Figure 29.

```
        A = DATA1(I-1,J)
        B = DATA1(I+1,J)
        C = DATA1(I,J-1)
        D = DATA1(I,J+1)

        DATA2(I,J) = 0.25 * ( A + B - C + D )
```

Figure 28 Variation 4 coding of Laplace example

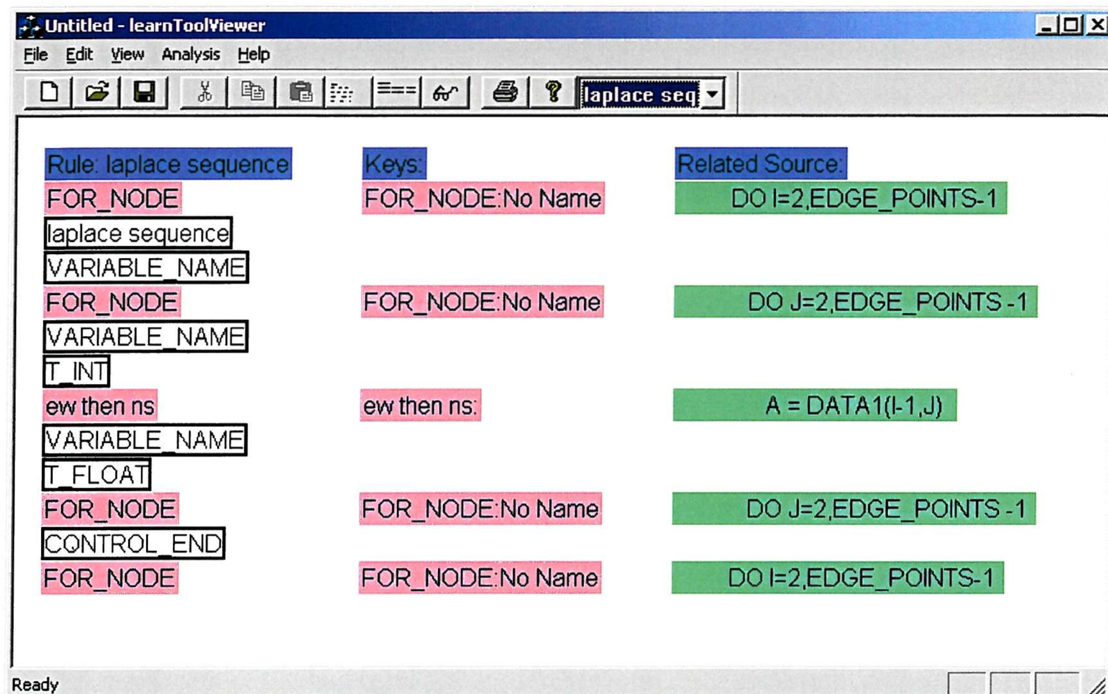


Figure 29 Screen shot of the match detail for the "laplace sequence" rule for this variation.

This property of incorrectly matching against a Rule, results from the Rule being defined to match without ensuring all the elements necessary for the higher-level concepts are present. This non-complete match approach is intended to mimic the behaviour of a human programmer, where the pattern matching procedure might start with what might be called a skim search. That is to say a first pass where a small set of key features are looked for. If a key feature were located during this skim search, then the programmer would form a hypothesis that this might be one of the concepts that is being looked for. Performing a more detailed search to locate the missing elements to tests the hypothesis would conclude the search process. Whilst a human programmer can make mistakes when the complexity of the formulation of a concept is high, the ability to ignore a lot of detail makes the searching very efficient, and

robust against the multitude of possible representations of the same concepts. The Rule formulation gains advantage from this flexibility as it allows matches to be attempted without needing extensive code normalisation before being applied. The down side is that the approach can make mistakes, like a human programmer.

3.3 Matrix Multiply Example

While no rules have been developed for this algorithm it is thought worthwhile to discuss how this approach would be expected to perform on this problem that has been described in some detail in Keßler [4] pp.78.

```
DO I=1,N
      DO J=1,N
         RET(J,I) = 0.0
            DO K = 1,N
               RET(J,I) = RET(J,I) + A(J,K) * B(K,I)
            END DO
      END DO
END DO
```

Figure 30 Code fragment for matrix multiply

Figure 30 shows a code fragment for the matrix multiply example translated into FORTRAN. The core matching is the multiply and add assignment with the specific array indices: (J,I) (J,K) and (K,I). The triple nest of the loop and the assign to zero would complete the rule construction. Unlike Keßler [4], Figure 30 shows the code with no loop unrolling. If the inner loop were unrolled the index pattern would be present for the first item of the unrolling and the match would probably be made. The problem would arise if the loop step on the inner loop was to be part of the match and the overall matching would fail since it had a non-unity step size.

In the absence of code normalisation transformations, it is probable that the concept would be recognised but the manipulation of the code would be error prone due to the way the match would ignore the un-wound elements of the source. The loop unwinding is an optimisation artefact and may cause human developers difficulty when trying to comprehend third party source code. This is a good example of where

code normalisation is beneficial and where premature optimisation can make source code difficult to maintain and re-optimise for a new platform.

3.4 Maximum Value in an Array of Integers

While no rules have been developed for this algorithm, it is thought worthwhile to discuss how this approach would be expected to perform on this problem that has been described in some detail in Paul et al [77] pp. 7.

```
MAX = ARRAY(1)
DO I=2,N
   IF ( MAX .LT. ARRAY(I) ) THEN
          MAX = ARRAY(I)
   END IF
END DO
```

Figure 31 Code fragment for find maximum integer value from array.

Figure 31 shows the code fragment for the find_max example. The first thing to notice is the need to have two core rules that are associated, namely "scalar less than array" and "array greater than scalar". These core rules would be combined together with the conditional assignment within the loop. The initialisation of the common scalar value would probably be assumed and a match would be made.

3.5 Discussion

With the limited number of examples examined it is difficult draw more than tentative conclusions from the work at this stage. The Reverse Traverse information as presented becomes very large for even small examples and the chosen presentation format is not ideal. In a completed system this information would perhaps be presented as an interactive tree much like a file system browser, allowing the user to limit the amount of data displayed at any one time. The Reverse Traverse output needs perhaps a prioritising metric so that if presented automatically, only the most important items are shown. There appears to be scope for research on how this metric might be formed and how successful the resulting filtering might be at guiding

the toolset user to the areas of code that would benefit most from the application of the other parallelisation tools in the toolset.

The pattern matching as implemented is fallible in certain instances. In some ways this similarity in performance is encouraging considering the starting point of the work. The trade off hoped for in terms of speed of recognition against accuracy is not proved. As implemented, in debug mode the matching process takes approximately 40 milliseconds to execute on the largest presented example on an 800 MHz Athlon PC. This is probably a respectable figure compared to the performance reported by Keßler [2] although for this example only about 20 rules were being compared and the example code length was a single subroutine.

The ability to match the rules developed to particular example codes has been shown, however the number and complexity of the rules is low and currently targeted at fairly simple algorithm components.

For a full system to be built, a means of systematically constructing rules would be required. This would involve identification of all the necessary information to extract from the parse tree to construct the codeStatusItems. The resolution of the ambiguous matching problem would also be necessary.

It may be that the deliberate avoidance of a code normalisation step is a fundamental flaw in the approach. It certainly makes formulating rules a non trivial task and perhaps makes it impossible to derive a systematic approach to developing general robust rules.

Chapter 4 Concluding Remarks

This work has presented a summary of existing approaches to the problem of developing and converting software to run parallel computer platforms. It has made the suggestion that algorithm substitution can be beneficial in some cases where the original serial algorithm is not well matched to a particular parallel platform and a change of algorithm can lead to significant improvements in execution speed and/or overall capacity. The work has also presented the results of the feasibility study to investigate the possibility of implementing such a system of transformations, which has lead to the construction of the Learn Tool program that incorporates the Reverse Traversal and pattern matching algorithms.

4.1 Parallelisation Tools

The extensive set of parallelisation tools developed over the last ten years highlights the need for support in programming parallel computer systems. Whilst there is support for code rewriting for distributing data and computation for SMP and MPP systems from sequential source, only a few tools are starting to consider algorithm substitution as a means of optimising the algorithm as well as the data and processing and distribution on these architectures.

4.2 Reverse Traversal

The reverse traversal concept is to start with a program and treat it as a "black box", determine its inputs and outputs and then open up the box and follow the processing between the two. By starting at the output and tracing backwards a focus is maintained on the data of most interest allowing a certain level of detail and potentially redundant code to be ignored. The main limitation of the algorithm is that it can only be reasonably beneficial within an interactive browsing facility. The Reverse Transverse tree expands very rapidly with relatively simple code examples and display of the whole tree is very rapidly unusable.

The Reverse Traversal could be used as a simple method of identifying the tightly nested loops of relevance in a practical situation and act as a trigger to enact the pattern matching on a subset of the code in a more developed tool.

4.3 Pattern Matching

The pattern matching approach has been shown to operate successfully on two examples. Its current limitations include the restriction of matching in a single subroutine at a time and the lack of confirmation checking after a possible match is located.

Both the pattern matching and the Reverse Traversal approach are based on the idea that the best way of implementing machine understanding and manipulation of source code should be based on approach adopted by human programmers. With pattern matching, the human programmer has a distinct advantage over machine approaches in the ability to pattern match in parallel (so to speak). Although the

author is un-aware of a definitive understanding of the human cognitive process, if neural network or memory surface concepts Bono [55] are indicative of these abilities, then humans can perform very rapid matches in parallel without the time consuming methodical comparisons used in this work. The human approach also allows for greater flexibility in the matching criteria, in that exact matches are rarely needed and multiple levels of detail are available in order to home-in on a match. The human programmer can also be prompted by source code comments to trigger a recognition or to prompt a more detailed search for the expected patterns in the code.

While computers have very powerful detail processing, for example comparing two strings, the number of detailed comparisons for all the possible variations of code representation can become very large for a relatively small number of patterns. This suggests that a less detailed and more parallel search method is required for an automatic system to address high complexity problems.

In order to introduce sufficient capability of variation in code representation, this work has allowed variable length gaps and associations between sub patterns. It has also needed to concentrate on the combination of small characteristic phrases in combination to infer that the higher-level pattern is in fact present during the Rule creation process. The Learn Tool approach could incorrectly recognise code examples that contain coding errors since there is currently no confirmation process whereby a possible match is then checked for completeness.

In other approaches Keßler [2][4], Pinter et al [3] and Keßler et al [5] code normalisation is used to reduce the variation between the templates and the examples. The approach of a human programmer is sometimes to normalise code, but often pattern matching is possible with no normalisation and it is for this reason that code normalisation was omitted from the approach.

4.4 Summary

This work describes two algorithms developed to test the feasibility of using program recognition techniques for program optimisation by algorithm substitution. The pattern matching algorithm has proved to be the best approach, a result that is in agreement with a number of authors Martino et al [1], Keßler et al [2], Pinter at al [3], Keßler [4][5], Raghavendra et al [6], Bansali et al [7], and Hagemeister et al [8].

The algorithms have been embodied in a program called Learn Tool, which includes a pattern creation editor and the matching algorithm scheme. This has been exercised successfully against a number of test code samples, the results of which are presented herein.

Appendix 1 – Example Source Code

A1.1 – Statistics

```
 1   C    This example is generated from Advanced Basic Scientific Routines
 2   C    B.V. Cordingley D.J Chamund page 50 MEAN AND VARIANCE.
 3
 4   C    The program opens a data file of choice. Filename prompted for
 5   C    and output mean and variance values to file called stats.out
 6
 7   C    3 Subroutines:
 8
 9   C    1. Prompt for filename, read file data into array
10   C    2. Calculate Mean and Variance
11   C    3. Output data to a file.
12
13   C    PERG 12-1-96 Started.
```

```
14    C=======================================================================
16          PROGRAM STATS
17
18          INTEGER MAX_ELEMENTS
19          PARAMETER (MAX_ELEMENTS=1000)
20          REAL ARRAY(MAX_ELEMENTS)
21          INTEGER NUM_ELEMENTS
22
23          CALL LOAD_DATA(ARRAY,MAX_ELEMENTS,NUM_ELEMENTS)
24
25          CALL CALC_STATS(ARRAY,NUM_ELEMENTS,MEAN,VARIANCE)
26
27          CALL OUTPUT_DATA(MEAN,VARIANCE)
28
29          END
30
```

```
31    C=====================================================================
32          SUBROUTINE LOAD_DATA(ARRAY,MAX_ELEMENTS,NUM_ELEMENTS)
33    C     ========================================
34
35    C     This routine prompts for an input file and reads column
36    C     orientated floating point data in free format.
37    C
38    C     e.g.    0.1786
39    C             5.6723
40    C
41    C     If there are more than MAX_ELEMENTS in the file the reading
42    C     is stoped short of the end.
43    C
44          REAL ARRAY(*)
45          INTEGER MAX_ELEMENTS
46          INTEGER NUM_ELEMENTS
47          LOGICAL EOF
48
49          CHARACTER *1024 filename
50
51            EOF = .FALSE.
52          NUM_ELEMENTS=0
53          WRITE(*,*)'Enter data file name'
54          READ(*,1)filename
55    1     FORMAT(A1024)
56
57          OPEN(UNIT=1,STATUS='OLD',FORM='FORMATTED',FILE=filename,ERR=10)
58            DO WHILE (.TRUE.)
59                  NUM_ELEMENTS=NUM_ELEMENTS+1
60                  READ(1,*,END=20)ARRAY(NUM_ELEMENTS)
61                  IF (NUM_ELEMENTS .EQ. MAX_ELEMENTS) RETURN
62          ENDDO
63
64          RETURN
65    10    WRITE(*,*)'Failed to open file:',filename
66          STOP
67    20    NUM_ELEMENTS=NUM_ELEMENTS-1
68          RETURN
69          END
```

```
 71   C=====================================================================
 72         SUBROUTINE CALC_STATS(ARRAY,NUM_ELEMENTS,MEAN,VARIANCE)
 73   C     ====================
 74
 75   C     This routine calculates the mean and variance of items in
 76   C     the array ARRAY. It looks at NUM_ELEMENTS of the array.
 77
 78
 79         REAL ARRAY(*)
 80         INTEGER NUM_ELEMENTS
 81         REAL MEAN
 82         REAL VARIANCE
 83
 84   C     Local variables
 85         REAL SUM1,SUM2
 86         INTEGER P
 87
 88         IF (NUM_ELEMENTS .GT. 2) THEN
 89                SUM1=0
 90                SUM2=0
 91                DO P=1,NUM_ELEMENTS
 92                       SUM1=SUM1+ARRAY(P)
 93                       SUM2=SUM2+ARRAY(P)*ARRAY(P)
 94                END DO
 95                MEAN=SUM1/NUM_ELEMENTS
 96                VARIANCE=(SUM2 - MEAN*SUM1)/NUM_ELEMENTS
 97   C            VRME=VARIANCE*NUM_ELEMENTS/(NUM_ELEMENTS-1)
 98   C            SDMK=SQRT(VRMK)
 99   C            SDME=SQRT(VRME)
100         END IF
101
102   C       Trap insufficient data occasions
103   C
104
105           IF (NUM_ELEMENT .LT. 2) THEN
106                WRITE(*,*)'Not Enough DATA to process'
107                MEAN=0.0
108                VARIANCE=0.0
109         END IF
110
111         RETURN
112
113         END
```

```
114    C=======================================================================
115          SUBROUTINE OUTPUT_DATA(MEAN,VARIANCE)
116    C     ======================
117
118    C     This subroutine outputs the MEAN and Variance to SDTOUT
119    C     and a results file called stats.out
120    C
121          REAL MEAN
122          REAL VARIANCE
125
126
127          OPEN(UNIT=1,FILE='stats.out',STATUS='UNKNOWN',
128         +       FORM='FORMATTED',ERR=10)
129
130          WRITE(1,*)'MEAN:',MEAN,' VARIANCE:',VARIANCE
131
132          CLOSE(UNIT=1)
133
134          RETURN
135
136    10    WRITE(*,*)'Error opening output file'
137          STOP
138
139          END
140    C=======================================================================
```

A1.2 – Laplace

```
 1  C*********************************************************************
 2  C
 3  C         PROGRAM: laplace1.ftn
 4  C         CREATED: 16:4:94 by Philip Galloway
 5  C         LAST CHANGE:
 6  C
 7  C
 8  C*********************************************************************
 9  C
10  C        This program performs a simple 2D Laplace solution on a
11  C        uniform square grid.
12  C
13  C*********************************************************************
14
15           PROGRAM laplace1
16
17  C        Define the number of cells along each edge.
18           INTEGER EDGE_POINTS
19           PARAMETER(EDGE_POINTS = 200)
20
21  C        Define the convergence limit
22           REAL CONVERGENCE_LIMIT
23           PARAMETER(CONVERGENCE_LIMIT=0.0001)
24
25  C        Declare the data stores.
26           REAL DATA1(EDGE_POINTS,EDGE_POINTS)  !First data store.
27           REAL DATA2(EDGE_POINTS,EDGE_POINTS)  !Second data store.
28
29  C        Set up the start data for iteration.
30           CALL INITIALISE_DATA_SPACE(DATA1,EDGE_POINTS)
31
32  C        Define the Boundary conditions.
33           CALL SET_BOUNDARY_CONDITIONS(DATA1,DATA2,EDGE_POINTS)
34
35  C        Iterate until convergence is reached.
36           CALL ITTERATE(DATA1,DATA2,EDGE_POINTS,CONVERGENCE_LIMIT)
37
38  C        Output the resultant data to a sequential binary file.
39           CALL OUTPUT_GRID_DATA(DATA1,EDGE_POINTS)
40
41           END
42
```

```
43  C ---------------------------------------------------------------------
44          SUBROUTINE INITIALISE_DATA_SPACE(DATA1,EDGE_POINTS)
45  C       ================================
46
47  C       This routine sets all the interior point data to zero.
48
49          INTEGER EDGE_POINTS !The dimension of the raw data space.
50          REAL DATA1(EDGE_POINTS,EDGE_POINTS)
51
52  C       Local array indexes.
53          INTEGER I,J
54
55  C       Set all interior points to zero.
56          DO I=2,EDGE_POINTS
57             DO J=2,EDGE_POINTS
58                DATA1(I,J) = 0
59                END DO
60             END DO
61
62  C       RETURN
63
64          END
65
```

```
 66   C -------------------------------------------------------------------
 67           SUBROUTINE SET_BOUNDARY_CONDITIONS(DATA1,DATA2,EDGE_POINTS)
 68   C       ==================================
 69
 70   C       This subroutine sets the edge boundary conditions using
 71   C       built in trig and polynomial functions.
 72   C       the full size edge is parameterised from 0-1 in which there
 73   C       are EDGE_POINTS samples.
 74
 75           INTEGER EDGE_POINTS          ! The dimensions of the stores.
 76           REAL DATA1(EDGE_POINTS,EDGE_POINTS)
 77           REAL DATA2(EDGE_POINTS,EDGE_POINTS)
 78
 79   C       LOCAL DATA
 80           INTEGER I,J                  ! Array indexes.
 81           REAL PI
 82           PARAMETER(PI=3.141596)
 83
 84
 85   C       Set up the I dependency functions
 86           DO I=1,EDGE_POINTS
 87
 88   C          Put the same BC's in both data stores.
 89              DATA1(I,1) = COS(2.0*PI*REAL(I)/REAL(EDGE_POINTS))
 90              DATA2(I,1) = DATA1(I,1)
 91
 92              DATA1(I,EDGE_POINTS) = SIN(PI*REAL(I)/
 93        +                               REAL(EDGE_POINTS))
 94              DATA2(I,EDGE_POINTS) = DATA1(I,EDGE_POINTS)
 95
 96              END DO
 97
 98   C       Set up the J dependency functions
 99           DO J=1,EDGE_POINTS
100
101              DATA1(1,J) = 0.1
102              DATA2(1,J) = DATA1(1,J)
103
104              DATA1(EDGE_POINTS,J) =0.9
105              DATA2(EDGE_POINTS,J) = DATA1(EDGE_POINTS,J)
106
107              END DO
108
109           RETURN
110
111           END
112
```

```
113   C -----------------------------------------------------------------------
114           SUBROUTINE ITTERATE(DATA1,DATA2,EDGE_POINTS,CONVERGENCE_LIMIT)
115   C       ==================
116
117   C       This subroutine iterates over the data until the CONVERGENCE_
118   C       LIMIT is no longer exceeded. Each call of the procedure
119   C       ITTERATE_STEP transfers the working data set from DATA1
120   C       to DATA2 or visa-versa. If the CONVERGENCE_LIMIT is not
121   C       exceeded at any point then DONE is returned TRUE.
122
123
124           INTEGER EDGE_POINTS           ! The dimensions of the stores.
125           REAL DATA1(EDGE_POINTS,EDGE_POINTS)
126           REAL DATA2(EDGE_POINTS,EDGE_POINTS)
127           REAL CONVERGENCE_LIMIT        ! The convergence limit.
128
129   C       Local variables.
130           LOGICAL DONE                  ! Detects the convergence situation
131           INTEGER STEPS                 ! Step counter.
132
133
134           DONE = .FALSE.
135           STEPS = 0
136           DO WHILE (.NOT. DONE)
137
138              CALL ITTERATE_STEP(DATA1,DATA2,EDGE_POINTS,
139        +                         CONVERGENCE_LIMIT,DONE)
140              STEPS = STEPS + 1
141              IF ( MOD(STEPS,10) .EQ. 0) write(*,*)STEPS
142              IF (.NOT. DONE) THEN
143                  CALL ITTERATE_STEP(DATA2,DATA1,EDGE_POINTS,
144        +                         CONVERGENCE_LIMIT,DONE)
145                  STEPS = STEPS + 1
146                  END IF
147
148              IF ( MOD(STEPS,10) .EQ. 0) write(*,*)STEPS
149              END DO
150
151          write(*,*)'Convergence took ',STEPS,' Steps.'
152
153          RETURN
154
155          END
156
157
```

```
158   C ---------------------------------------------------------------------
159           SUBROUTINE ITTERATE_STEP(DATA1,DATA2,EDGE_POINTS,
160       +                          CONVERGENCE_LIMIT,DONE)
161   C       ======================
162
163   C       This subroutine preforms a single itteration step using
164   C       the Jacobi method for solving the Laplace Equation
165   C       on a regular square grid.
166
167           INTEGER EDGE_POINTS         ! The dimensions of the stores.
168           REAL DATA1(EDGE_POINTS,EDGE_POINTS)
169           REAL DATA2(EDGE_POINTS,EDGE_POINTS)
170           REAL CONVERGENCE_LIMIT      ! The convergence limit.
171           LOGICAL DONE                ! Finish after convergence reached.
172
173   C       Local data
174           INTEGER I,J                 ! data array indexes.
175           REAL CURRENT_DIFF           ! Current convergence size
176           REAL MAXIMUM_DIFF           ! Maximum convergence size
177
178           MAXIMUM_DIFF = 0.0
179           DO I=2,EDGE_POINTS-1
180
181              DO J=2,EDGE_POINTS -1
182
183                 DATA2(I,J) = 0.25*( DATA1(I-1,J) + DATA1(I+1,J) +
184       +                             DATA1(I,J-1) + DATA1(I,J+1) )
185
186   C             Find the convergence test at this point.
187                 CURRENT_DIFF = ABS(DATA2(I,J) - DATA1(I,J))
188                 MAXIMUM_DIFF = MAX(MAXIMUM_DIFF,CURRENT_DIFF)
189
190                 END DO
191
192              END DO
193
194           IF (MAXIMUM_DIFF .LT. CONVERGENCE_LIMIT) THEN
195               DONE = .TRUE.
196               END IF
197
198           RETURN
199
200           END
201
202
```

```
203  C ---------------------------------------------------------------------
204          SUBROUTINE OUTPUT_GRID_DATA(DATA1,EDGE_POINTS)
205  C       ===========================
206
207  C       This routine opens a binary sequential file and
208  C       dump the data information in DATA1
209
210          INTEGER EDGE_POINTS
211          REAL DATA1(EDGE_POINTS,EDGE_POINTS)
212
213          INTEGER I,J     ! array indexes.
214
215  C       Open the file
217          OPEN(UNIT=10,FORM='UNFORMATTED',ACCESS='SEQUENTIAL',
218      +        FILE='Laplace.bin',STATUS='UNKNOWN')
219
220          DO I=1,EDGE_POINTS
221                 WRITE(10) (DATA1(I,J),J=1,EDGE_POINTS)
222                 END DO
223
224          CLOSE(UNIT=10,STATUS='KEEP')
225
226          RETURN
228          END
```

Bibliography

[1] Beniamino Di Martino and Christoph W. Keßler, 'Program Comprehension Engines for Automatic parallelization: A Comparative Study', Software Engineering for Parallel and Distributed Systmes, pp144-157, Chapman & Hall, 1996.

[2] C. W. Keßler, W. J. Paul, 'Automatic parallelization by Pattern Matching', Proc. Of Second Int. Conference of the Austrian Centre for Parallel Computation, Springer LNCS 734, pp 166-181, October 1993.

[3] S. S. Pinter, R.Y. Pinter, 'Program Optimization and Parallelization Using Idioms', ACM SIGPLAN, Principles of Programming Languages, pp 79-92, 1991

[4] C. W. Keßler, 'Pattern-driven automatic program transformation and parallelization', Proc. Euromicro workshop on Parallel and Distributed Processing 1995, pp76-83

[5] C. W. Keßler, C. H Smith,' The SPARMAT approach to automatic comprehension of sparse matrix computations', Proc. 7th International workshop on program comprehension 1999, pp 200-207

[6] C. S. Raghavendra, S. Bhansali, 'On porting sequential programs to parallel machines', Proc. COMPSAC'94 pp 313-318

[7] S. Bansali, J. R. Hagemeister, et. al, 'Parallelizing sequential programs by algorithm-level transformations' Proc. IEEE Third workshop on program comprehension 1994, pp100-107

[8] J. R. Hagemeister, S. Bansali, et. al., 'Implementation of a pattern –matching approach for identifying algorithmic comcepts in scientific FORTRAN programs', Proc. 3rd Int. conference on High Performance Computing 1996, pp209-214

[9] Kelly T, 'Optimising Hardware Granularity in Parallel Systems', PhD University of Edinburgh 1995

[10] Sabot G., Wholey S. et al, 'Parallel Execution of a FORTRAN 77 Weather Prediction Model', Proc 1993 ACM/IEEE conference on Supercomputing, Portland Oregon, USA.

[11] Sethian J. A., 'Computational Fluid Mechanics and Massively Parallel Processors' , University of California, Proc. Supercomputing 1993.

[12] Jin H., Hirbar M. and Yan J., 'Parallelization of ARC3D with Computer-Aided Tools', NAS Technical Report , NAS-98-005, NASA Ames Research Center 1998.

[13] Hong Q Ding, 'Monte Carlo Simulation of Quantum Systems on Massively Parallel Supercomputers', Proc. Supercomputing 1993.

[14] Ewing R.E. Sharpley R.C., 'Distributed Computation of Wave Propagation Models using PVM', Proc. Supercomputing 1993.

[15] Cross M., Ierotheou C. S., Johnson S.P., Legget P. and Rvans E., ' Software tools for Automating the parallelisation of FORTRAN Computational

Mechanics Codes', Parallel and Distributed Processing for Computational Mechanics 1997.

[16] Harman M.,Simpson S.,'EPSILON - A Radar Cross Section Modelling package', Proc. UKSCS 1990

[17] Galloway P., Simpson S., 'Parallelisation of Radar Cross Section (RCS) Prediction for Electrically Large Targets', Proc, European Simulation Multiconference 1991

[18] Galloway P.,'The Operation and Control of large Parallel Simulations on a Network of Sun Workstations using CSTools™',Proc. European Simulation Symposium 1992

[19] Knapik M, Johnson J., 'Developing Intelligent agents for distributed systems', McGraw-Hill, 1998.

[20] Haridi S., Van Roy P. et. al., 'Programming Languages for distributed applications', New Generation computing, 16(3):223-261, May 1998.

[21] Wolfe M., 'Data Dependence and Program Restructuring', Jou. Supercomputing 4 (321-344) 1990

[22] Wolfe M., 'Engineering a data dependence test', CONCURRENCY: Practice and Experience, Vol 5(7), 603-622 Oct 1993

[23] Brooks, E. D. III and Warren K. H. 'A Study of Performance on SMP and Distributed Memory Architectures Using A Shared Memory Programming Model'. Proc. of SuperComputing (Nov 1997).

[24] Pountain D., May D.,'A tutorial Introduction to occam Programming'.,BSP Professional Books.

[25] Foster I., Taylor S.,"Strand: New concepts in parallel programming", Prentice Hall 1990. ISBN 013850587X

[26] Chapman B., Mehrotra P., et al, 'Dynamic Data Distributions in Vienna Fortran', Supercomputing 1993 Pp. 284-293

[27] Fox G., Hiranandani S., et al., 'Fortran D language specification.' Dep. Computer Science Rice COMP TR900079. 1991.

[28] Harris J. et. al.,'Compiling High Performance Fortran for Distributed-memory Systems',Digital Technical Journal Vol. 7 No. 3 1995

[29] Brandes T., Benkner S., 'Exploiting Data Locality in Scaleable Shared Memory Machines with Data Parallel Programs', Proceedings of Euro-Par 2000, Parallel Processing. September 2000.

[30] OpenMP Fortran/C Application Program Interface, http://www.openmp.org/

[31] Jin H.M, Frumkin M., Yan J., 'The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance', NAS Technical Report, NAS-99-011, NASA Ames Research Centre, 1999.

[32] Silicon Graphics, Inc. MIPSpro ™ Power Fortran 77 Programmer's Guide. Document 007-2361-007, SGI, 1999.

[33] Thinking Machines Corporation, 'CM Fortran Reference Manual', version 5.2, Cambridge, MS, 1989

[34] Su E., Lain A., et.al., 'Advanced Compilation Techniques in the PARADIGM Compiler for Distributed Memory Multicomputers', Proc. ACM International Conference on Supercomputing, Barcelona Spain, 1995.

[35] Walker E., 'Extracting data flow information for Parallelizing FORTRAN nested loop kernels', PhD thesis, University of York (U.K.) June 1994.

[36] Das R., Saltz J., et al ' Slicing Analysis and Indirection Access to Distributed Arrays' Proc. Sixth Annual Workshop on languages and Compilers for Parallel Computing, Portland OR. August 1993.

[37] Van der Wijngaart R. F., 'Charon toolkit for parallel, implicit structured-grid computations: Literature Survey and conceptual design' NAS Report 97-018, NASA Ames Research Centre, Moffett Field, CA., 1997

[38] Saad Y., Kuznetsov S., et al. 'PSPARSLIB: A portable library of parallel sparse iterative solvers', Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997.

[39] Schonauer W., Hafner H., Weiss R., 'LINSOL, a parallel iterative linear solver package of generalised CG-type for sparse matrices', Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997.

[40] Message Passing Interface, http://www-unix.mcs.anl.gov/mpi

[41] Snir M., Otto S. W., et. al, 'MPI: The Complete Reference', MIT Press, 1995

[42] Meiko Ltd., 'CS TOOLS for MeikOS', Meiko Almondsbury, Bristol England

[43] Saad Y., Sosonkina M., 'Non-standard parallel solution strategies for distributed sparse linear systems', Proceedings of ACPC'99, Lecture notes in computer science, Berlin, 1999. Springer-Verlag. http://citeseer.nj.nec.com/308666.html.

[44] Folors N. Reeve J., 'Domain Decomposition Tool (DDT) version 2.2 An abridged Users Guide', Southampton University 1994, ESPRIT CAMAS 6756.

[45] Adams M. F., 'A Distributed Memory Unstructured Gauss-Seidel Algorithm for Multigrid Smoothers',ACM/IEEE proceesinds of SC01: High Performance Networking and Computing 2001., http://citeseer.nj.nec.com/adams01distributed.html

[46] Charles Simonyi, "The Death of Computer Languages, The birth of Intentional Programming', Technical Report, MSR-TR-95-52, Microsoft Corporation.

[47] Advanced Parallel Research, 'Forge Explorer User's Guide', http://www.qpsf.edu.au/workshop/forge/forge.html

[48] J.H. Merlin. 'Inter-procedural Dependency Analyser (IDA)' CAMAS report 2.2.1.1, Univ. of Southampton March 1993.

[49] Bodin F., Beckman P. et al, 'Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools', University of Rennes and Indiana University. 1994, OONSKI94

[50] 'Sage++: A Class library for Building FORTRAN 90 and C++ Restructuring Tools:User Guide', Nov 1994, Indiana University.

[51] KAP: http://www.kai.com/vkomp

[52] Allen F.,Burke M., et al, 'An Overview of the PTRAN Analysis System for Multiprocessing',Jou. of Parallel and Distributed Computing 5 617-640 1988.

[53] Zima H. P.,Bat H. J.,Gerndt M., 'SUPERB: A tool for semi-automatic MIMD/SIMD parallelization', Parallel Computing 6 (1988) 1-18 North-Holland.

[54] Cross M., Ierotheou C. S., et al,'CAPTools - semiautomatic parallisation of mesh based computational mechanics codes' Pres. to HPCN Europe Munich 1994. (April).

[55] Simulog http://www.simulog.fr

[56] Hood, R, ' Building a Portable Distributed Debugger', proc. STDT'96: SIGMETRICS Symposium on Parallel and distributed Tools. 1996.

[57] May J., Berman F., ' Panorama: A Portable, Extensible Parallel Debugger', ACM SIGPLAN, 28(12), December 1993.

[58] Nikolaou, C., Saridakis, T., Zarras A, 'ArrayTracer : A Parallel Performance Analysis Tool', Technical Report TR95-0136, Aug 1996 http://citeseer.nj.nec.com/article/nikolaou96arraytracer.html

[59] London, K., Dongarra, J. et al. ' Using PAPI for hardware performance monitoring on Linux Systems', Presented at Linux Clusters : Revolution, July 2001., http://www.ptools.org/projects/index.html

[60] Shende S., Hackstadt S. T. and Malony A. D., ' Dynamic performance callstack sampling: Merging TAU and DAQV-II', Proceedings of the Fourth International Workshop on Applied Parallel Computing (PARA98), June

1998., Lecture notes in Computer Science, No. 1541, Springer-Verlag, Berlin 1998.

[61] 'Animated Algorithms', The Mathematica Journal vol4, issue4 Fall 1994 pp37- 43, based on work by Roman E Meader, ETH Zurich, Institute of Theoretical Computer Science.

[62] Fitzpatrick s. et. al, 'Deriving Efficient parallel Implementations of Algorithms Operating on General Sparse matrices using Automatic Program Transformations'. Parallel Processing CONPAR 94-VAPP VI pp 148-159.

[63] ten Cate H.H., Vollebregt E.A.H, "On the portability and efficiency of parallel algorithms and software" Report 94-76, NOWESP project within MAST II program.

[64] R. C. Waters. 'Automatic analysis of the logical structure of programs.' Technical Report 492, MIT Artificial Intelligence Lab., December 1978.PhD Thesis.

[65] S.F Fickas and R Brooks. 'Recognition in a program understanding system', In Proc. 6th International Joint conference. Artificial Intelligence, pages 2660268, Tokyo, Japan, August 1979.

[66] L. M. Wills. 'Automated Program Recognition by Graph Parsing', PhD thesis, MIT 1992, Technical Report 1358, MIT Artificial Intelligence Lab, Cambridge, MA. http://users.ece.gatech.edu/~linda/phd-thesis.html

[67] W. Kozaczynski, J.Ning, et. al. ,'Program concept recognition and Transformation', IEEE Transactions on Software Engineering, 1992. pp1065-1074

[68] C. Rich., 'A formal representation of plans in the programmer's apprentice.', Proceedings of the 7th International joint conference on artificial intelligence. 1981.

[69] L. Snyder. 'Recognition and Selection of Idioms for Code Optimization', Acta Informatica, 17:327-348, 1982.

[70] Stroustrup B., 'The C++ Programming language', Third Edition, Addison-Wesley, 1997. ISBN 0-201-88954-4

[71] C. G. Nevill-Manning, I. H. Witten, ' Identifying Hierarchical Structure in Sequences: A linear Time algorithm', http://dna.stanford.edu/sequitur/jair

[72] Villavicencio G., 'Program Analysis for the Automatic Detection of Programming Plans Applying Slicing', Proceedings of the 5th European Conference on Software Maintenance and Reengineering. CMSR01, March 2001, http://citeseer.nj.nec.com/villavicencio01program.html

[73] Kucherov G., 'Matching a Set of Strings with Variable Length Don't Cares', Theoretical Computer Science, 178 (1997) pp 129-154.

[74] Baker B. S., Giancarlo R., 'Longest Common Subsequence from Fragments via Sparse Dynamic Programming', European Symposium on Algorithms, August 1998.

[75] Baker B. S., et al. 'On Finding Duplication and Near-Duplication in Large Software Systems', Second working conference on Reverse Engineering, 1995.

[76] Baker B. S., et al., 'Compressing Differences of Executable code', ACM SIGPLAN workshop on compiler support for system software (WCSSS'99), 1999.

[77] Paul, S., Prakash A, "A Framework for source code search using program patterns", IEEE Transactions on Software Engineering 20(6): 463-475.

[78] Quilici A., Yang Q., et al., 'Applying Plan Recognition Algorithms to Program Understanding', Automated Software Engineering: An International Journal, July 1997, http:// citeseer.nj.nec.com/quilici97applying.html

[79] Teukolsky S. A. et al. 'Numerical recipes in C, The art of scientific computing, Second Edition', Cambridge University Press, ISBN 0-521-43108-5, 1992

[80] Bono Edward de, ' The Mechanism of Mind', Penguine Books Ltd, ISBN 0-1402.1445-3, 1969.