Automatic conversion of programs from serial to parallel using Genetic Programming - The Paragen System

Paul Walsh
University College
Cork
paul@csvax1.ucc.ie

Conor Ryan
University College
Cork
conor@ravenloft.ucc.ie

Abstract

This paper describes a novel system of converting sequential programs into functionally equivalent parallel programs without using data dependency rules. This system utilises the powerful directed search technique of Genetic Programming to find the most efficient program in terms of parallelism. A brief description of the field of Genetic Programming is given, followed by a discussion on the design and implementation issues of this system.

1 Introduction

One of the major restrictions of the use of parallel hardware is the problem of producing efficient software for general non SIMD type architectures and relevant problems. This is due in part to the architectural complexity and diversity of parallel hardware. While there are a large number of parallel processing platforms available, there is often a lack of software design expertise and tools for these platforms.

Moreover, there is no easy way to port the software written for a serial (or parallel) platform to a parallel platform. Clearly, there is a need for efficient translators to address this problem. Such tools could perform translation, in total, or they could offer the programmer a facility for exploiting low level parallelism in certain sections of code. Furthermore, translation tools could also allow the developement of new applications in traditional serial languages, which can then be translated to the language of the target machine [Banarjee 93].

The current thinking on the problem of autoparallelisation advocates the use of data dependencey analysis and program transformation techniques. However, such approaches have limitations due in part to the lack of sophistication of current autoparallelisation tools [Blume 94]. Indeed, the problem of autoparallelisation is non-trivial and the number of rules for the required symbolic analysis of programs is significant. Also the application of transformations to sequential code requires considerable processing.

This paper introduces the *Paragen* system, a system which employs Genetic Programming [Koza 92] to allow the automatic conversion of serial programs. Paragen does not employ

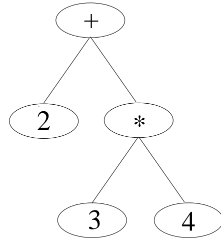

Figure 1 : A simple parse tree

any data dependency rules or analysis of any kind, either algorithmically or on the part of the programmer, thus considerably reducing the effort required for parallelising code.

Using the Genetic Programming paradigm, Paragen effectively evolves parallel programs which are functionally equivalent to an orginal, serial program.

2 Genetic Programming

Genetic Programming(GP) is a relatively new technique for the generation of computer programs. GP, like its ancestor, the Genetic Algorithm(GA), harnesses the power of evolution to solve problems. Starting with an initial population of randomly created programs, "individuals", GP literally evolves a solution to the problem at hand.

Like natural evolution, the evolution that occurs in GP relies both on the genetic structure of the individuals that are under-going evolution, which allows the production of new individuals, and on some sort of selection pressure which makes it imperative for the individuals to constantly improve or face extinction.

2.1 Representation in Genetic Programming

Unlike traditional GA, GP uses program trees to represent individuals. So, for example the LISP program **(+ 2 (* 3 4))** could be represented as in Figure 1. These program trees are made up of two fundamental building blocks, nodes and leaves. Nodes are simply functions such as + * which take one or more arguments, while the leaves are terminals, i.e. numbers or zero-argument functions. The first major step in any implementation of GP is to correctly identify the necessary functions and terminals and to ensure that any combination of them well result in a syntactically (although not necessarily functionally) correct program.

2.2 Survival of the fittest

An initial, random population of these individuals is created. Each of these individuals is then evaluated, testing how suited they are to the particular task at hand. Typically, there are a number of test-cases from which each individual is assigned a score.

The next step in GP is to create the next generation in the population. Individuals are selected probabilistically from the previous (*parent*) generation to be either *reproduced* or *crossed over*. An individual chosen to be reproduced is copied unchanged into the next

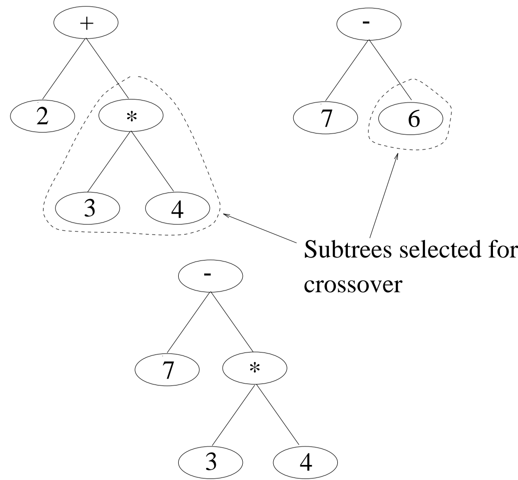

Figure 2 : Crossing over two parent trees by swapping sub-trees

generation. If an individual is chosen for crossover, a second individual is also chosen, and these are then crossed over, as described in the next section, to produce a new individual.

2.3 Crossover

Due to the nature of the representation scheme used by GP, it is possible to swap sub-trees from two individuals. This results in the creation of two new, syntactically correct individuals, see Figure 2. Crossover is the exploration tool of GP. It is by crossing over good performing individuals the population as a whole slowly evolves until a "perfect" individual appears, i.e. an individual who fulfills all the requirements of a problem. Many individuals, who will be produced as the result of a crossover, do not perform well, and, in true Darwinian evolutionary style, they will be selected neither for crossover or reproduction in the following generation, and so die off.

3 The Paragen System

The Paragen System was designed to be able to take a serial program, and automatically transform it into a functionally equivalent one. Individuals in the Paragen system are parse trees which map a program onto a parallel machine. Four functions, all of the form **XN**, were made available to the system, **P2, PN, S2** and **S3**. **X** denotes whether the subtrees are to be evaluated in **P**arallel or **S**erial. **N** denotes the number of subtrees attached to this node. Figure 3 shows some example programs.

The next decision was to isolate what terminals were to be made available. Given that the functions tell Paragen how to map something onto a parallel machine, it was decided that the terminals should tell Paragen what to map onto the machine. This means that each statement or line of code in the original serial program becomes a terminal.

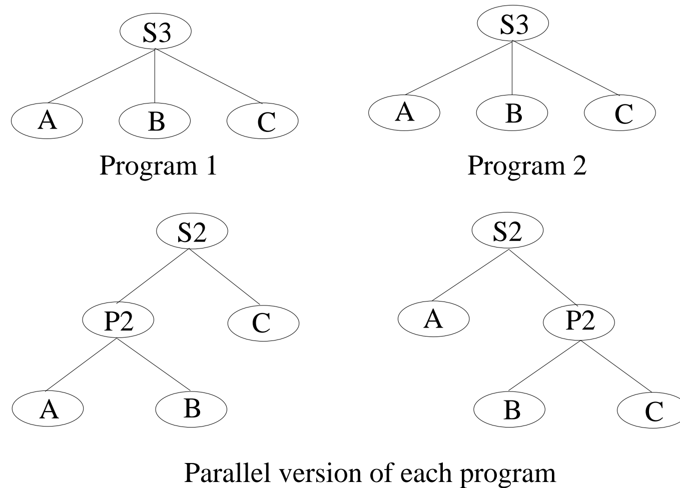

Figure 3 : Two example programs, shown in serial and parallel form

```
Stmt A : a = b + 2
Stmt B : c = b + 5
Stmt C : c = e + f
```

```
Stmt A : a = a + b
Stmt B : b = a + 1
Stmt C : c = a + 3
```

Two example sequential programs, shown in parse tree form in Figure 3

3.1 Functions and Terminals

Unlike virtually every other parallelisation technique, Paragen makes no attempt whatsoever to analyse the original program. In fact, Paragen completely disassembles a program into its constituent statements, and attempts to rebuild it in a parallel form using these statements and a combination of the parallel and serial functions available.

3.2 Testing individuals

The first step testing an individual in Paragen is to instantiate it on a (virtual) parallel machine. Individuals are then subjected to a number of test-cases, each of which involves different starting values for the variables used. The program is then executed on the parallel machine and the resulting values compared to the those values produced by the serial program. Each time the parallel version gets the same result as the serial version, its score is incremented.

3.3 Selection in Paragen

Most problems tackled by GP involve solving a set problem, such as sorting numbers, recognising a shape, performing regression etc., but the problem of autoparallellisation is not quite so straightforward. The problem faced by Paragen is a two-pronged one.

To maintain a balanced population of individuals who are both correct and parallel, Paragen employs the *Pygmy Algorithm*[Ryan 94a]. The Pygmy Algorithm differs from normal

GP selection in that there are *two* parent populations. Individuals in the first population are scored primarily on their correctness, but also take parallelism into consideration, while those in the second population are scored primarily for their parallelism, with some consideration being given to correctness. Each parent population is made up of a list of individuals, sorted on their score.

When an individual is tested, it is compared against the individuals in each of the parent populations. If the individual has scored high enough to enter the population, the lowest member of that population is removed. This approach of keeping a list of parents is known as a *Steady-State* approach and means that reproduction is no longer needed, as high performing individuals are always preserved.

3.4 The Perfect Individual

One difficulty with any GP system is knowing when to terminate a run. This difficulty arises because it is often impossible to identify when a perfect individual has appeared, although there is no difficulty in identifying which individual is the best from any given run. There are all manner of approaches to terminating a run prematurely, based on lack of change in the population, or on an individual attaining a certain score.

In this paper, we are more concerned with showing that the Paragen system is capable of autoparallelisation than with optimizing GP, so we simply allow the run to terminate after an arbitrarily set number of generations.

4 Paragen - The First Results

This paper serves very much as an introduction to the workings of Paragen. To illustrate how Paragen operates, it has been tested on a variety of common parallelisation problems from the literature[Bruanl 93]. For these experiments, GP was run with populations varying from 20 to 80 individuals, for 20 to 50 generations, depending on the difficulty of the problem.

4.1 Autoparallelisation Problems - Data Dependencies

Given a sequential program it is often necessary to determine the order of execution of the statements within that program. Often there is a dependency between the statements in the sequential program where the order of execution must be preserved. Statements without any dependencies can execute in parallel. With current autoparallelisation techniques these dependencies must be determined before the correct transformation can be carried out. In the Paragen system, however, these dependencies are automatically determined by the fitness function. Initially, the Paragen system was sucessfully tested with individual problems characteristic of the three major types of data dependency, namely Flow-dependency, anti-dependency and output dependency. However, for more realistic problems, we tested the system with loops that contain various types of dependencies, as described in the following section.

4.2 Loops

The parallelisation of loops is one of the most important aspects of autoparallelisation, as the bulk of computing workloads are controlled by loop type structures. In parallelising loops, there may be data dependencies both within the loop itself and across the different loop iterations. Data dependencies that span loop iterations are known as cross iteration dependancies. This is especially significant as many parallelisation techniques attempt to execute the different iterations of a loop in parallel, but ignore any parallelism within the loop. This approach alleviates the data dependancy constraint within the loop as often the original sequence of statements is preserved. However with the flexibility of the Paragen system we can simultaneously extract parallelism both within the loop and across loop iterations.

Another consideration in the parallelisation of loops is array access. Loop control structures are often used in the processing of arrays and this further complicates data dependency analysis techniques. The parallelisation of loop iterations in the Paragen system is made possible by the introduction of another function, the **DoAcross** function. **DoAcross** takes three arguments, the initial value of the index, the number of iterations and the loops to be executed. A corresponding sequential loop was also made available, **Do**.

In the following example, which is an extended version of the problem described in [Bruanl 93], there is a data dependency across different loop iterations due to the statements S4 and S8. Here statement S4 of one iteration must be executed before statement S8 of the next iteration, due to the flow dependency caused by e[i-1]. This requires a synchronisation mechanism to be used between different loop iterations. However, because the Paragen system also converts the statements within the loop iteration, by re-ordering and / or parallelisation, the effect of the synchronisation mechanism on performance is minimised. In the case of a generated code being run on a machine that requires synchronisation, a simple post-processing can be applied to the program to determine where synchronisation mechanisms.

Our extensions to this problem are the introduction of a number of dependencies within the loop, between S1 and S2, between S6 and S7 and between S5 and S6.

```
for i := 1 to n do
  begin
    S1: a[i] := f[i] * 2;
    S2: f[i] := 57;
    S3: g[i] := d[i] + 8;
    S4: e[i] := e[i] + d[i];
    S5: b[i] := a[i] + d[i];
    S6: a[i] := b[i] + 2;
    S7: a[i] := d[i];
    S8: c[i] := e[i-1] * 2;
  end;
```

The resulting code produced by Paragen:

```
    DoAcross i := 1 to n
      begin
```

```
    PAR-BEGIN
      S1: a[i] := f[i] * 2;
      S3: g[i] := d[i] + 8;
      S4: e[i] := e[i] + d[i];
    PAR-END
    PAR-BEGIN
      S8: c[i] := e[i-1] * 2;
      S5: b[i] := a[i] + d[i];
    PAR-END
    PAR-BEGIN
      S2: f[i] := 57;
      S6: a[i] := b[i] + 2;
    PAR-END
    S7: a[i] := d[i];
end;
```

4.3 Paragen and Synchronisation

In the virtual machine that Paragen uses, all processors execute at exactly the same speed, so there is no need for synchronisation between shared variables being used at different time steps. If an asynchronous machine is to be used, some synchronisation may be necessary. This can be done by post-processing of the generated parallel program to detect any flow dependencies, the only dependencies that may affect the running of the program, and synchronisation mechanisms can be automatically inserted.

4.4 Paragen and loopholes

As stated earlier, the fitness of a program is a combination of how correct it is and how parallel it is, but Paragen (and GP in general) proved to be quite adept at discovering loopholes. A common strategy, especially in the shorter examples, was for Paragen to execute all the instructions in parallel, regardless of dependencies. This gave the individual a high score in terms of parallelism - the program executing in only one time step - and a high score in terms of correctness was achieved by ensuring that each statement was executed so many times that the variables affected ended up with the correct values. This behaviour was curbed by reducing an individual's fitness each time it repeated an instruction.

Paragen also noticed that in the case of output dependencies, where statement S2 writes to the same variable as S1, S1 could often be left out of the final program as it didn't affect any final values. Clearly, this is not correct, so it was necessary to reduce an individuals fitness each time it left out an instruction.

Another strategy that had to be discouraged was that of individuals who didn't do anything. These individuals exploited the fact that several variables used by many programs don't change, and by not doing anything these individuals would get a score by virtue of the fact that these variables will always contain the correct value at the end of the run. To prevent individuals such as these from having a chance of being selected, only those variables which

appear on the left hand side of statements are considered when calculating an individual's score.

5 Proving Paragen

Currently the programs produced by Paragen can be proved to be functionally identical to the original program by writing the converted program in CSP notation. One of the major benefits in converting the code to CSP is that the resulting program may be formally verified.

6 Conclusion and Future Directions

The Paragen system is a new auto-parallelisation tool which can be used to convert serial programs without any analysis for dependencies. Paragen has been successfully tested with code containing a number of data dependencies, and has successfully converted loops containing cross-iteration dependencies. As well as converting entire loops, Paragen can also determine sections of code within those loops that can be parallelised while still preserving any dependencies contained within.

Extending Paragen to convert entire functions or subroutines is the next obvious step, and, from the extremely encouraging results presented in this paper, should not be too difficult.

The next step for Paragen is to produce code in the Occam language, using channels as necessary for synchronisation, which would allow the generated code to be used on a wide variety of parallel machines.

The eventual goal for Paragen is to create parallel programs on the fly, with just a statement of the problem which is to be solved. This would eliminate the need to write the original, serial program, which would further automate the process of generating parallel code.

References

[Banarjee 93] Banerjee, U. et al, Automatic Program Parallelization, Proceedings of the IEEE, Vol. 81, No. 2, February 1993.

[Blume 94] Blume W., et al. Automatic Detection of Parallelism, IEEE Parallel and Distributed Technology, Fall 1994.

[Bruanl 93] Braunl, T. Automatic Parallelisation and Vectorization, Parallel Programming an Introduction, Prentice Hall, 1993, ISBN 0-13-336827-0.

[Burns 88] Burns, A. (1988) : Transforming Occam Programs. In Programming in Occam2 : Addison-Wesley, 1988, ISBN 0-201-17371-9.

[Koza 92] Koza, J. (1992) : *Genetic Programming.* Cambridge : M.I.T. Press.

[Ryan 94a] Ryan, C. (1994) : Pygmies and Civil Servants. In Advances in Genetic Programming. Ed. K. Kinnear Jr. Cambridge : MIT Press