Conor Ryan and Laur Ivan

This chapter describes a Genetic Programming system, Paragen, which transforms serial programs into functionally identical parallel programs. Unlike most other GP systems, it is possible to prove that the programs generated by the system are functionally identical. The ability to prove that the output of a GP run is correct has greatly improved the chances of GP being used in a commercial situation.

2.1 Introduction

Until recently, parallel programming tended to be restricted to either purely academic activities or to exotic super computer systems which were normally the preserve of wealthy institutions. The advent of systems such as PVM[Geist, 1993] (Parallel Virtual Machine)/ MPI (Message Passing Interface) and Linda[Gelernter, 1985] have changed this, however, by treating a network of (possibly heterogeneous) computers as though each were a node in a parallel computer.

The performance and practicality of these systems has further improved with the use of Beowulf systems, which are generally groups of Intel or Alpha-based machines on a fast (100MBit or greater) local network running a version of PVM or MPI. These systems have all the characteristics of the PVM type systems mentioned above, with the added advantage of extremely fast communication, thus allowing the possibility of increasingly fine grains of execution.

Parallel processing is becoming increasingly important as more and more sophisticated techniques are being developed for areas such as simulations, engineering applications or graphics rendering.

2.1.1 Software Re-engineering

Despite the apparent ease with which one can adopt parallel architectures, they have yet to enjoy widespread use. One important reason for this is that the kind of users who stand to benefit most from parallel processing tend to have large legacy systems running on serial machines. Re-writing this legacy code can represent an enormous cost.

Software re-engineering, the re-writing of code in a different form while retaining its functionality, represents a significant investment, as demonstrated by the existence of re-engineering companies the sole service of whom is to provide Year 2000 solutions[Piercom]. The most successful re-engineering companies are those that have developed tools that automate, or at the very least, semi-automate, the re-engineering process. In general, the greater the level of automation, the greater the success rate and thus, the less testing required.

Due to the difficulties associated with re-writing existing serial code, many organizations are not in a position to take advantage of these attractive new architectures. Many organizations have neither the resources nor expertise required to produce parallel code, and often, those that do, are faced with the problem that the quality of the code is directly related to the expertise of the programmer involved.

There are currently no automatic parallelization tools available. Parallel compilers such as HPF, KAP Fortran etc. can generate parallel code, and, in some cases, (KAP Fortran) identify standard simple transformations, they were designed to take advantage of code that was written with the intention of being executed in parallel, rather than to convert serial code.

Difficulties with the production of good parallel code are not restricted to re-writing, however. The generation of parallel code is an arduous task that requires a substantial mind-shift and no small amount of training, particularly if the code is to be optimized. Persons or institutions wishing to produce parallel code would stand to benefit from a tool that would allow them to develop their code in a traditional manner, and subsequently convert it to parallel code. Of course, programmers who take this route would, by neccesity, demand proof that the newly converted code is equivalent to their original code.

We have developed a software re-engineering tool, Paragen, that adheres to the above demand, to the extent that both re-engineering and testing are fully automated. We have found that GP has proved particularly suitable for the generation of parallel code, because it eagerly embraces the maze of transformations required for re-engineering. Furthermore, the often lateral approach required for parallel program design, while an athema to many programmers raised on imperative mind-set, is tailor made for the bottom up approach GP takes.

2.2 Parallel Problems

Current techniques for auto-parallelization rely heavily on data dependency analysis techniques. This consists of analyzing the statements of a program to determine if there is any data dependency between them. If there is no chain of dependence between two statements then they can execute in parallel. For example, to analyze whether two statements S1 and S2 are independent then the set of used variables S1.U and S2.U, and the set of modified variables S1.M and S2.M must be determined. Then, if

$$S1.M \cap S2.U = \epsilon$$

and

$$S1.U \cap S2.M = \epsilon$$

and

$$S1.M \cap S2.M = \epsilon$$

then statements S1 and S2 are independent[Braunl, 1993]. These data dependencies are known as flow dependencies, anti-dependencies and output dependencies respectively. Note that it is possible for $S1.U \cap S2.U \neq \epsilon$ to hold, but this is reflective of the case where both instructions read the same variable, and as neither modifies it, there is no dependency.

With current auto-parallelization techniques these dependencies must be determined before the correct transformation can be carried out. With Paragen, however, these dependencies can be automatically determined, and subsequently punished, by the fitness function.

2.2.1 Problems with Data Dependency Analysis

Auto-parallelization techniques that rely on data dependency analysis as outlined above have limitations due to both the complexity of discovering which data dependency rules can be used. Further limitations are caused by the difficulty associated with the subsequent determination of the order in which to apply those transformations [Lewis, 1992]. In general, if analysis shows that n transformations can be applied to a program, $n!$ different programs can result, depending on the order in which the transformations are applied. Finding the most suitable order of application is, as yet, an unsolved problem.

This problem, however, is particularly suitable for GP, as it can be used to find at least a near optimal order for these transformations. Moreover, not only can GP discover the order, it can also be used to determine which transformations may be legally applied.

Generally, when parallelizing a program, transformations are performed according to a set of rules, of the form

$$SEQ(A,B) = PAR(A,B)$$

if all the dependency rules from above hold. These rules are associative, so

$$SEQ(A,B,C) = SEQ(A,SEQ(B,C))$$

Clearly, if there are no dependencies between the instructions B and C, but if there were some dependency between A and B, then one could say

$$SEQ(B,C) = PAR(B,C)$$

By substitution we get

$$SEQ(A,B,C) = SEQ(A,PAR(B,C))$$

There are two main difficulties associated with data dependency analysis, the identification of which transformation rules can be applied legally, and the order in which to apply them[Lewis, 1992]. If a system could discover the rules and the order of application, it would not only be able to parallelize a program, but also record exactly how the program was parallelized. Proof of equivalence of such a program would then be a simple matter, as the rules prescribe which statements can be analyzed for dependency.

Paragen significantly reduces the complexities associated with data dependency analysis by utilizing the power of Genetic Programming to direct the programmer to areas of the program that are most likely to benefit from analysis.

2.3 Genetic Structure

Unlike most applications employing GP, Paragen doesn't evolve programs, rather it evolves sequences of *transformations*. The system can be viewed as an *embryonic* one, in that one starts with a serial program, and progressive application of the transformations modify it, until eventually the system produces a parallel version of the program. It is only after this embryonic stage that an individual is tested.

All the transformations employed are standard, and syntax preserving with one caveat; that is, that the area of the program which a transformation affects does not contain any data dependencies. If a transformation violates that condition while being applied it may change the semantics, and any transformation that runs this risk causes an individual's fitness to be reduced.

When calculating an individual's fitness, all the transformations are examined to test if they have caused any dependency clashes. The speed of the program produced by an individual is also calculated, which is simply the number of time-steps it takes to run the program.

The first class of transformation are designed to cater for blocks of sequential code, and are based on standard Occam-like transformation rules. These are rules of the form

$$SEQ(A\,B) = PAR(A\,B)$$

which state that two instructions, A and B, can be executed in parallel if there are no dependencies between them.

However, by far the greatest parallelism can be extracted from loops, as it tends to be within these that the largest amount of processing is carried out. Again, there are a large amount of standard transformations, such as loop skewing, fusion etc.

To reflect the dual nature of the transformations, individuals must be capable of employing both, and applying the appropriate type depending on the nature of the code encountered in the embryonic stage. Each type of transformation is stored separately, the *atom* type, which affect single instructions, are manipulated by standard tree structures in a similar fashion to standard GP. The second type of transformation, the *loop* type are stored in a linear genome.

The reason for the different type of representation can be seen from the example below. An individual is evaluated in “atom mode”, that is, the tree is traversed in the normal manner, with the transformations being applied to the serial program. However, if a loop structure is encountered, the system enters “loop mode” and a transformation is instead read from the linear part of the genome, which is then applied. After this, the system returns to atom mode and applies the next available transformation to the code within the loops, thus, not only is it possible to parallelize the loop itself, but also the code contained within the loop.

Separate genomes are required because the loop transformations may only be applied to loops, and a linear genome holds these transformations as all that is required is a sequential list to determine the order in which the transformations should be applied. However, due to the nature of some of the loop transformations which give the property that several transformations may be applied to the same loop (or group of loops) the genome is divided into several sections, each containing a number of possible transformations for each loop.

To permit the system to change seamlessly between the two modes, we view the entire program as atoms. Ordinary instructions are atoms in the normal sense, in that they cannot be broken down further, while loops are known as *Meta-loop* atoms, which consist of one or more adjacent loops.

2.3.1 Atom Mode

Unlike all other implementations of GP, individuals in Paragen are evaluated in normal order[Peyton-Jones, 1992], that is, individuals are evaluated from the outermost level. Usually, GP systems employ applicative order, where individuals are evaluated from the inside out, or, in terms of trees, from the bottom up.

Normal order, also known as *lazy evaluation* or *call by need*, attempts to delay the evaluation of arguments to a function by evaluating the function before the arguments. The arguments are evaluated only if strictly necessary.

Consider the rather unlikely situation

$$(\lambda x.\ 1) < bomb >$$

An applicative order GP will first evaluate the argument, i.e. $< bomb >$. In the case where $< bomb >$ leads to a non-terminating state, e.g. infinite recursion, disaster will follow. However, normal order will not evaluate $< bomb >$ until it is needed, and elegantly avoids any catastrophe because the expression does not examine its argument, instead returning the value 1, regardless of what it is called with.

Consider the expression (using lambda calculus for clarity)

$$(\lambda\ xy.(+\ 3(*\ x\ x)))(+\ 4\ 5)(*\ 3\ 2)$$

This situation is similar to an expressions $(+\ 4\ 5)$ and $(*\ 3\ 2)$ being passed to an ADF $(\lambda(xy)...)$. Using applicative order as is the norm for GP, the following execution sequence results:

$$(\lambda\ xy.(+\ 3(*\ x\ x)))9\ (*\ 3\ 2)$$

$$(\lambda\ xy.(+\ 3(*\ x\ x)))9\ 6$$

$$(+\ 3\ (*\ 9\ 9)$$

$$(+\ 3\ 81)$$

Notice how the $(*3\ \ 2)$ is evaluated regardless of the fact that the lambda expression doesn't actually need it. The sequence would be quite different for normal order:

$$(\lambda\ xy.(+\ 3\ (*\ x\ x)))(+\ 4\ 5)(*\ 3\ 2)$$

$$(+\ 3(*\ (+\ 4\ 5)(+\ 4\ 5)))$$

The next step would be to apply the outer $+$, but this is a strict function which must have both of its arguments evaluated, so instead the $*$ operator is applied. Again, this is a strict function, so only now are the two $(+\ 4\ 5)$ evaluated.

In this case $(+\ \ 4\ \ 5)$ is evaluated twice, while $(*\ \ 3\ \ 2)$ is not evaluated because it is not needed. Normal order is often used to reduce the number of evaluations, but, as this example demonstrates, is not always successful because the number of evaluations is zero *or more*. Paragen is not concerned with reducing evaluations, rather it exploits the order in which operators are applied when employing normal order.

The simplest transformations in Paragen perform a transformation on the current state of the program, and then execute their argument which is passed the modified state of the program. These functions are described further on.

However, there are other instructions, namely `P50` and `S50`, which take two arguments. These functions divide the program into two separate parts, and apply one argument to each half. Consider the individual in figure 2.1, the `P50` divides the program segment P into two parts P' and P''. The lower levels of the tree, `A` and `B` are applied to P' and P'' respectively, *after* their parent node has been applied.

These transformations effectively fork the execution of an individual, permitting it to reflect the parallel nature of the program it is modifying.

Each transformation in atom mode operates on the current *program segment*. Typically, a transformation schedules one or more atoms relative to the rest of the program segment, before passing the rest of the segment onto the next transformation(s). The segments get

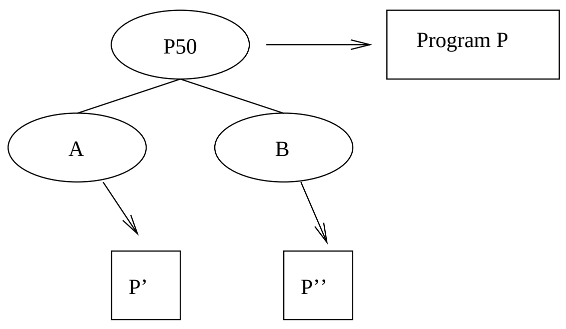

Figure 2.1
An example of normal order evaluation in Paragen. The transformation P50 is applied to all the program, but A and B are only applied to P' and P" respectively.

increasingly smaller until there are either no atoms left, or all the transformations for a particular segment are exhausted.

The crucial result of using Normal order is that the tree structure of individuals indicates which transformations should be applied to various parts of a program, and, moreover, chains of transformations can be built up, performing all manner of modifications to a program.

2.3.2 Atom Mode Transformations.

All atom mode transformations operate in the same manner. Before any application, the systems checks to see if the first atom in the current program segment is a meta-loop atom. If so, the system enters loop mode, otherwise the transformation is applied to the segment.

There are four categories of atom mode specific transformations defined:

1. *Pxx*/*Sxx*,

2. *Fxxx*/*Lxxx*,

3. *SHIFT*,

4. *NULL/PARNULL*.

2.3.2.1 P and S

The Pxx/Sxx transformations are the most general of the transformations. These break the current program segment into two new segments, by putting a certain number of the atoms into each segment. The proportion of atoms that go into each segment is determined by the 'xx' part, which is a percentage. Table 2.1 and figure 2.2 show an example of P20 being applied to `[ABCDE]`.

Table 2.1
The operation of the P20 transformation

Operation	Input chain	Output
P20	`[ABCDE]`	`[A]`
		`[BCDE]`

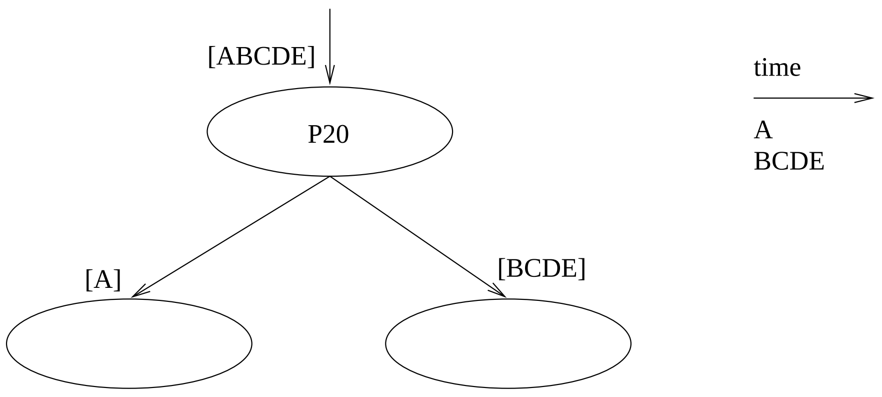

Figure 2.2
A fork of execution caused by the application of the P20 transformation.

The [A] and [BCDE] will be executed in parallel and [A] will be passed on to the left subtree and [BCDE] to the right subtree. When called, the fitness function will determine if there is a data dependency between [A] and [BCDE]. If there is no data dependency between the sequences, then both will be executed in parallel. Otherwise, the information will be used by the fitness function as described later on. Notice that there are now two program segments, one corresponding to each of the two groups of atoms generated by the transformation.

Consider the ‘Sxx’ operator, with xx=60. This causes the two new segments to be executed in sequence, an operation which preserves their original order. However, this can be of use if some parallelism can be extracted from a smaller segment.

Again, the transformation generates two program segments, each of which can have more transformations applied to them. However, the order specified by the first transformation will always be adhered to, because all scheduling of atoms is done relative to other atoms

Table 2.2
The operation of the S60 transformation

Operation	Input section (program)	Output (result)
S60	`[ABCDE]`	`[ABC] [DE]`

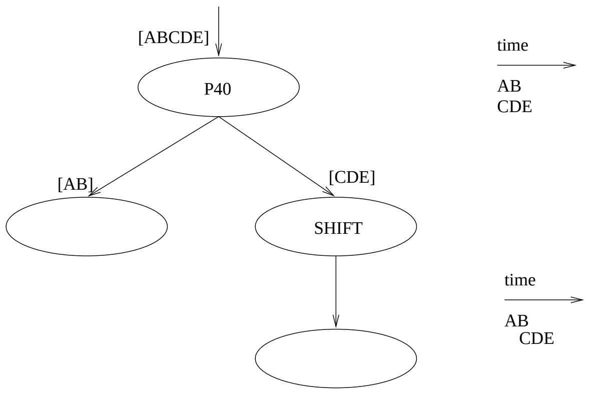

Figure 2.3
The application of SHIFT operator to a program segment, the execution of the entire segment is delayed by one time-step.

in the current program segment.

2.3.2.2 F and L

These classes can be looked upon as extreme cases of the Pxx/Sxx transformations. The difference in this case is that only a single (either **F**irst or **L**ast) instruction is scheduled, which is quite helpful when the system needs to trim down a program segment. A consequence of there only being a single instruction scheduled is an increase in the fineness of the granularity of the code.

Each class has two transformations, namely FPAR/LPAR and FSEQ/LSEQ, all of which remove one statement from the current program segment and passes the remainder of the segment onto the next transformation.

The F transformations remove the first atom, executing the remainder in either parallel or in sequence, while the L transformations remove the final atom. Again, this atom is either executed in parallel or in sequence, depending on the nature of the transformation.

2.3.2.3 SHIFT

SHIFT is a simple transformation which doesn't alter the order of a program segment, but delays the execution of all the atoms it contains by one time step.

If there are other atoms to be executed (in sequence) after the segment shifted, then they too, are shifted, otherwise a data dependency clash could be introduced, which would be very difficult to detect.

Shift is necessary as it helps other transformations fine tune their effects. For example, if P40 was the only operator, and there is a data dependency between A and C, the individual would be penalized when evaluating the fitness, because P40 states that A and C are to be executed in the same time.

However, if the right branch of P40 contains a SHIFT operator as in figure 2.3 , then the execution of the [CDE] sequence is delayed with one time step. This means C will start to be executed once A is finished. The result is equivalent to:

$$\texttt{SEQ([A],PAR([B],[CDE]))} \tag{2.1}$$

With this addition, the individual will not be penalized anymore for the A-C data dependency, because A will be executed before C.

2.3.2.4 NULL/PARNULL

The final class of transformations do not take any arguments, and are used to terminate a tree. **NULL** causes all remaining instructions in the current program segment to be executed in their original, sequential order, while **PARNULL** causes any remaining instructions to be executed in parallel.

NULL and PARNULL always appear on the leaves of an individual as they do not take any arguments.

2.3.3 Atom Mode Fitness

The most important factor to be considered for the fitness function is the *correctness of the program*. Clearly, Paragen would quite happily generate massively parallel programs which could run in times as quick as one time step at the cost of correctness.

Only transformations which alter the order of atoms, i.e. FPAR, LPAR, Pxx, need be tested, as it is in these cases that dependency clashes may appear.

Each (parallel) transformation rule is responsible for ensuring that the modifications it makes to the program do not violate any data dependencies. Starting with the most deeply nested rule, each performs any necessary checking on the current program segment. After each is finished checking, it recursively passes the current segment back to the previous rule. Below are detailed algorithms for the checking required by some of the parallel transformations. In all cases, A represents the instruction(s) affected by the operations, and B the instructions remaining in the program segment. We denote the original time step of execution(T_i) of instruction n of group A as A_{ni} and the new time step as A_{nj}. In some cases there will only be one instruction in group A, but for consistency the same notation will be used throughout.

2.3.3.1 Directed Analysis for FPAR

FPAR takes the statement that was originally first in the segment and executes it in parallel with the segment (after the segment has been modified by zero or more subsequent operations). A possible data dependency violation can occur if the first statement is being executed at the same time step as a statement that must occur after it. This is detected by testing all possibilities of a violation:

```
For all instructions $B_i$ in $B$
    If $A_{0j} == B_{nj}$
    check $A_0$ and $B_i$ for dependency.
```

If there is a dependency, the individual is punished, but the checking continues.

2.3.3.2 Directed Analysis for LPAR

LPAR takes the statement that was originally last in the segment and executes it in parallel with the segment, again, the remainder of the segment may be modified. There is far more scope of dependency violation in this case, as LPAR is effectively bringing the execution time of the statement forward. In this case, all other statements are examined, as they will all be executed either at the same time or after the statement that was moved.

2.3.3.3 Directed Analysis for Pxx

Pxx divides a segment in two, and executes the result of each segment in parallel. In this case, the execution time setup of several statements are effectively changed, so this operation requires the most analysis. Similar to FPAR above, each statement in the *group* A, is compared with each statement in the group B. Any change in the order of execution of two statements is then examined for dependency violations.

```
For each statement $x$ in group $A$
    For each statement $y$ in group $B$
        If ( ($A_{xi} < B_{xi}$ ) && ($A_{xj} >= B_{xj}$) )
        Check
```

2.3.3.4 Directed Analysis

As can be seen from the above sections, the directed analysis section requires information about when all the instructions are going to be executed. However, when evaluating an individual in normal order, this information isn't yet available, because subsequent transformations are likely to change some of the execution times.

To avoid any incorrect information being passed to the analysis stage, we wait until the entire individual has been evaluated, and then evaluate it a second time, this evaluation uses bottom up, applicative order as normally employed in Genetic Programming.

The bottom up approach of the second evaluation permits us to examine the simplest transformations (i.e. those involving the fewest instructions) first, and pass any information about dependency clashes back up the tree. An example of this is given in section 2.4.

2.3.4 Loop Mode

Paragen remains in atom mode until it encounters a program segment which starts with a meta-loop, whereupon it switches into loop mode. While in loop mode, Paragen reads from the linear genome, which contains one gene for each loop encountered while parsing the original program. Each gene is made up of one or more loop transformations, all of which are applied to the meta-loop.

It is important to have a number of transformations for each meta-loop, as some transformations, e.g. Loop Fusion in section 2.3.4.1 have certain requirements that must be fulfilled before they can be applied. In order to fulfill these requirements it is often necessary to "massage" other loops around, or to ensure that the loops contained within the meta-loop are already parallel.

Loop optimization is crucial for auto-parallelization as the greatest amount of processing tends to be executed within these structures. The body of loops are subject to the same dependencies as other code, but also suffer from the possibility of *cross-iteration* dependencies. These are dependencies which cross over two or more iterations:

```
a[i]=x;
y=a[i-1];
```

Fortunately, there are all manner of modifications and alterations which can be carried out on loops to encourage greater parallelism. We have identified a number of these from which we have generated transformations which are made available to Paragen. Below are a representative sample of some of the more interesting ones.

2.3.4.1 Loop Fusion

Loop Fusion is a loop specific transformation which selectively merges two loops into a single loop [Lewis, 1992]. We use the term "selectively" because two requirements must be filled before loop fusion can be applied, they are that both loops must be already parallelized, and that both loops must contain the same number of iterations.

Consider the following loops:

```
PAR-FOR statement1; END
PAR-FOR statement2; END
```

After applying the Loop fusion operator, the result is:

```
PAR-FOR statement1; statement2; END
```

However, it is relatively unusual for two consecutive loops in a program to meet the two above criteria. If either or both of the loops is not already parallel, Paragen will attempt to parallelize them first. A success in this endeavor will permit the transformation to continue.

Another situation is both loops are of the same type, but the iteration domains are different. There are two ways to approach this:

- if one number of iterations is multiple of the other number, e.g.

```
PAR-FOR i=1 TO 100 statement1; END
PAR-FOR i=1 TO  10 statement2; END
```

then there are several options:

- generate an inner loop

  ```
  PAR-FOR j=1 TO 10
    PAR-FOR k=1 TO 10 statement1(k+10*j); END
    statement2(j);
  END
  ```

- conditioning statement 2 execution within an *if*

  ```
  PAR-FOR j=1 TO 100 statement1;
    IF((j MOD 10)==0)
      //j  is divisible by 10
    statement2(j/10);
  END
  ```

- unrolling the bigger loop

  ```
  PAR-FOR j=1 TO 10
    statement1(0+10*j);  statement1(1+10*j);
    statement1(2+10*j);  statement1(3+10*j);
    statement1(4+10*j);  statement1(5+10*j);
    statement1(6+10*j);  statement1(7+10*j);
    statement1(8+10*j);  statement1(9+10*j);
    statement2(j);
  END
  ```

  Notice that the instructions that are created as a result of the loop being unrolled can be further parallelized.

- for un-normalized loops (and indefinite number of iterations)

```
PAR-FOR i=alpha TO beta statement1; END
PAR-FOR i=gamma TO delta statement2; END
```

the solution is:

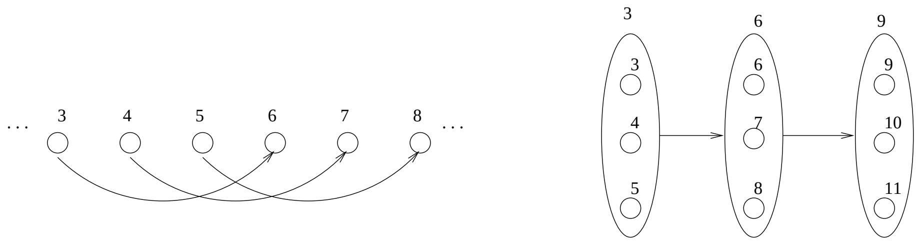

Figure 2.4
The operation of the Shrink transformation. Instructions between the cross-iteration dependencies are grouped together, ensuring that communication only takes place after all instructions in a group are executed.

```
lim1=MAX(alpha,gamma)
lim2=MIN(beta,delta)
PAR-FOR i=alpha TO lim1 statement1; END
PAR-FOR i=gamma TO lim1 statement2; END
PAR-FOR i=lim1 TO lim2 statement2; END
PAR-FOR i=lim2 TO beta statement2; END
PAR-FOR i=lim2 TO delta statement2; END
```

Notice that the two initial loops can be normalized in order to have the same start index or the same end index. Also, note that from the previous statement, one of the first two loops will not be executed. If *lim1 = alpha* then the first loop will not be executed. Otherwise the second loop will not be executed. The same reasoning is applied for the last two loops. In the end, there will be maximum of three loops, and two if we can shift the domain for one of them.

This transformation is quite representative of some of the more elaborate loop transformations in Paragen's repertoire, in that it relies on other transformations to be applied first. Several of Paragen's transformations behave in this way, and it is for this reason that each meta-loop can have several transformations applied to it.

2.3.4.2 Loop Shrinking

Loop Shrinking is a transformation for parallelizing interlaced cross-iteration data dependency loops.

When all dependences in a cycle are flow dependent, there are no direct transformations for obtaining a good result. However, depending on the distance of each dependence, parts of the loops can be parallelized using loop shrinking [Lewis, 1992].

Figure 2.4 presents the input and the result after applying the shrinking operator.

Given the source:

Table 2.3
The view of the program at the start of the evaluation of an individual.

Instruction No.	1	2	3	4-12	13	14-18	19	20
Atom Name	A	B	C	D	E	F	G	H
Pseudo timestep	0	1	2	3	4	5	6	7

```
FOR i=4 TO n  a[i]=a[i-3]+x[i]; END
```

then the operator transforms it in:

```
FOR i=4 TO n STEP 3
  PAR-FOR j=i TO i+2 a[j]=a[j-3]+x[j]; END
END
```

Note that, at the end, the inner loop is already parallelized, not “waiting to be parallelized”. The drawback of this operator is that if there is no possible parallelization, then the individual will be penalized. If the loop is already parallel, it will still try to generate a nested loop based on the parallel source, and the individual will not be penalized.

2.4 Example Individual

We now examine an the execution of an example individual taken from a population. The code we are concerned with is illustrated in figure 2.5, and consists of twenty lines of code, containing four loops.

Notice how the loops are divided into two groups, one of three loops and the other containing two (nested) loops. This division is governed by the adjacency of the loops, i.e. the loops containing instructions 4 - 12 are contiguous, and are thus treated as a single meta-loop. This gives eight distinct sections to this program, namely, instructions 1,2,3,4-12, 13, 14-18, 19 and 20.

Furthermore, as we are currently concerned with the execution of these instructions *relative* to each other, we give each statement a *pseudo-timestep*. Clearly, this abstraction will have to be addressed by a scheduler after the experiment if the code is to be optimized, but it is necessary at this stage to ensure that the correct data dependency analysis is performed.

The individual is described in figure 2.4, and consists of a tree with seven nodes. Three of these are atom transformations, while the remaining four are the NULL transformation, used to terminate atom mode.

Before a Paragen run, the program is parsed to extract certain information, namely the set of variables modified and used (read) by each statement. Thus, a table similar to figure 8 is constructed.

This table will subsequently be used to test the transformations for any data dependency clashes. As stated earlier, the individual is executed in normal order, so the first transformation to be applied is the `S63`, which is applied to the entire program. The only test

```
 1.  a[5]=a[4]+b[3];
 2.  counter++;
 3.  index--;
 4.  for(i=1;i<N;i++) {
 5.      c[i]=x+y+i;
 6.  }
 7.  for (i=1;i<M;i++) {                4-12
 8.      b[i]=m[i]+x+i;
 9.  }
10.  for (i=1;i<N+1;i++) {
11.      a[i]=x+a[i]+1;
12.  }
13.  b[3]=b[0]+a[0];
14.  for (i=2;i<N;i++) {
15.      for(j=2;j<N;j++) {
16.          d[i,j]=(d[i+1,j]*d[i,j+1])/index;     14-18
17.      }
18.  }
19.  counter*=index;
20.  a[b[3]]=a[3];
```

Figure 2.5
Serial code to be modifi ed, notice how adjacent loops are treated as a single metaloop atom.

carried out at this stage is to see if the first block of code is a loop or atom. In this case, it is an atom, so the transformation can successfully be applied. S63 states that the first 63% of instructions must be executed before the remaining 37%. Clearly, this is an order preserving transformation which will not require any dependency checking. Execution of the individual is now forked, with instructions A-E being passed to the left hand subtree, while instructions F-H are catered for by the right subtree. Regardless of what transformations are subsequently applied, the first five groups will always execute before the remaining three.

The next transformation is P60, which executes the first 60% of its group of instructions in parallel with the remaining 40%, giving a situation as in figure 2.9. In this case, there are two major changes to the program. The group [D E] is now being executed at time step 0, while the group [F G H] is to be executed at time step 3, which is still after the original [A B C D E F] group. In a similar manner to S63, the first half of the group is sent to the left transformation, while the second goes to the right hand one.

The left hand subtree contains the NULL transformation, which merely examines the

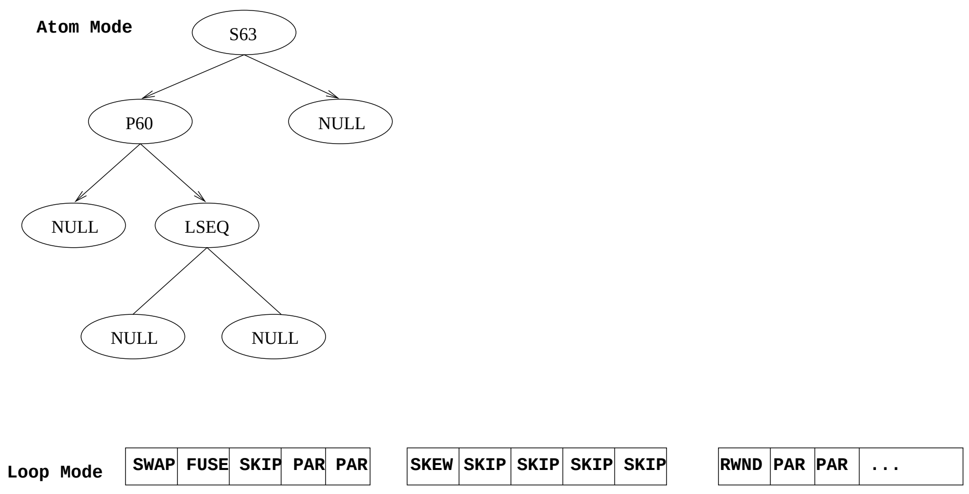

Figure 2.6
An example individual from Paragen. The familiar tree structure is used for processing atomic instructions, while there is also a set of linear genes, one for each metaloop.

	Modified	Used
1.	a	a
2.	counter	counter
3.	index	index
4-12.	a, b	m, x, a, y
13	b	b, a
15-18.	c	c, index
19.	counter	counter, index
20.	a	b, a

Figure 2.7
The sets of used and modifi ed variables, notice how the loops are treated as compound statements.

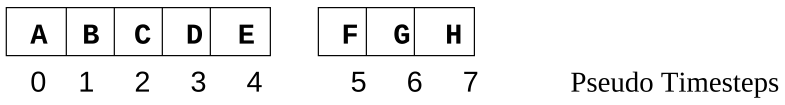

Figure 2.8
The state of the program after applying the S63 transformation.

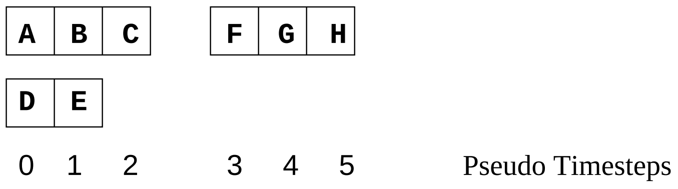

Figure 2.9
The state of the program after applying the P60 transformation.

group to test if it begins with a loop. In this case it doesn't, so there will be no further transformations applied to [A B C]. The right hand `LSEQ` transformation, however, does encounter a loop, and causes Paragen to enter loop mode.

Once Paragen exits loop mode, `LSEQ` is applied to that program segment, and causes the final atom to be executed after the meta-loop atom.

2.4.1 Loop Mode

Once into loop mode, Paragen is only concerned with the first instruction in the current segment, which will always be a meta-loop. The meta-loop is expanded, in this case into three separate loops, which we denote X, Y and Z:

```
X :for (i=1;i<N;i++)
       { c[i]=x+y+i; }
Y: for (i=1;i<M;i++)
       { b[i]=m[i]+x+i; }
Z: for (i=1;i<N+1;i++)
       { a[i]=x+a[i]+1;  }
```

The first (loop-specific) transformation to be applied to this new [X Y Z] segment is `SWAP`, which inverts the first two loops, yielding [Y X Z]. The second transformation is `FUSE` which, as described in section 4.4.1 joins the common parts of X and Z together, giving a program segment which we can describe as [Y XZ z'], where z' is the part of the Z loop which is exceeds the index of the loop in X.

Notice that the loop transformations do not reduce the size of the program segment. Rather they are progressively applied to the loops within the segment. Thus, if there are more transformations left after we have gone through all the loops, we wrap the program segment and continue, which means that each loop can have one or more transformations applied to it. In this case, however, there is a new loop, z', to which the next transformation, SKIP, is applied. SKIP is analogous to the NULL transformation for atom mode, and does not modify a loop.

Since the loops in the segment are exhausted, we wrap it, and continue with the application of our transformations, starting with Y. This transformation is PAR which simply converts the loop into a PAR-FOR, which maps the loop across up to the index number of processors. The final transformation is also a PAR-FOR, and is applied to the newly created XZ loop. At this stage, the transformations are exhausted, so the system reverts to atom mode.

2.4.2 Resumption of Atom Mode

When all the transformations are exhausted, Paragen returns to atom mode. As the next transformation is a NULL, there is no more to be done with this part of the tree, so execution continues with the right hand side of the tree. Eventually, the program below is generated.

```
PAR
  BEGIN
    a[5]=a[4]+b[3];
    counter++;
    index--;
  END
  BEGIN
    PAR-FOR (i=1;i<M;i++) {
       b[i]=m[i]+x+i;}
    PAR-FOR (i=1;i<N;i++) {
       c[i]=x+y+i;
       a[i]=x+a[i]+1; }
    for (i=N;i<N+1;i++) {
       a[i]=x+a[i]+1;}
    b[3]=b[0]+a[0];
  END
END-PAR
PAR-FOR (i=2;i<N;i++) {
  for (j=i+2;j<=i+N-1;j++) {
    d[i,j-1]=(d[i+1,j-i]*d[i,j-i])/index; }
    }
counter*=index;
a[b[3]]=a[3];
```

Testing the speed of this program is a trivial task, however, testing the correctness of the program is another matter, and requires analysis of all the transformations which modified the order of the atoms of the program. Notice that in this particular example, there still exist some areas in which parallelism can be extracted, most notably the first three statements, which are all independent of each other.

2.4.3 Directed Data Dependency Analysis

The final step in Paragen is to determine the number (if any) of data dependency clashes in the parallel program. To do this, the individual is evaluated a second time, this time in the traditional GP manner, from the bottom up. Starting with the most deeply nested transformation, the program segments associated with each of the leaves is examined for possible clashes, checked, and the result passed back up the tree.

The program segment associated with the deepest transformation, i.e. the NULL on the left of LSEQ, is first examined. This consists of just the atom D, a metaloop atom, which may have been subject to some transformation. However, all the system was forced into loop mode by the LSEQ transformation, and not the NULL, so the analysis is delayed until that node in the tree is reached.

After this test, the system examines the right hand side of the LSEQ. This program segment contains a single atom, which is not a loop, and thus requires no further analysis. LSEQ itself is an order preserving transformation, so the atoms D and E can be reassembled without further checking.

Once this program segment ([D E]) exists again, it is checked to see if it begins with a meta-loop, which it does. This causes the metaloop gene to be called a second time, and on this occasion it checks for any dependency clashes that may have occurred. This checking is relatively straightforward, as all the transformations are standard, taken in the most part from sources such as [Burns, 1988] and [Lewis, 1992]. Furthermore, once a particular transformation has been applied to a particular loop, the question of whether or not any dependency clashes have been introduced can be answered immediately if required by any subsequent individuals.

Working back up the tree, we now examine the leaves of P60 transformation, starting with the left subtree which was applied to [A B C]. When the NULL of P60 is applied to that segment, there was no further transformation, as the first atom, namely A is not a meta-loop.

The right hand side has already been checked, as this was a level deeper. Therefore, we now check the effect of P60 itself, and, as none of the first three atoms employ any variables used by the second two, both segments can successfully be executed in parallel.

Taking another step back up the tree, we encounter S63, another order preserving transformation. In this case, it doesn't matter if the segments it created are dependent, because they don't interfere with each other. However, a check is still made on the NULL in its right hand subtree, to investigate the possibility of the presence of a meta-loop atom.

Investigation reveals that there is indeed a meta-loop atom at the head of that program segment, and so the requisite analysis is performed on that atom. Again, as this is a standard transformation, the analysis is quite straightforward.

2.4.4 Experimental Results

Paragen is currently being tested on benchmark code to get an idea of how well it performs relative to other techniques. So far, the code has always performed at least as well as human experts at extracting parallelism, and always does so much more quickly. Furthermore, to our knowledge there are no other fully automatic parallelization techniques available.

The speed up obtained from code depends on the code itself, and there is no simple rule of thumb for determining the level of parallelization likely to be extracted. Code that contains no dependencies, clearly, can achieve greater than code with heavily dependent instructions. Furthermore, code that contains several loops offers a much greater pay-back than code that is simply made up of atomic instructions.

Systems such as High Performance Fortran etc. are unlikely to extract too much parallelism from most code that was written with sequential execution in mind. If such compilers do detect parallelism, it simply good fortune, as they are designed for programs written specifically for parallel architectures. Paragen, on the other hand, actively seeks out possible transformations, and even takes adjacent instructions into account.

Consider the code below :

```
0 : a=1;
1 : for (int i=0;i<100;i++)
      b[i]=a;
2 : a=10;
3 : for (int i=0;i<100;i++)
      c[i]=a;
4 : for (int i=0;i<200;i++)
      d[i]=a;
5 : a=100;
6 : f=a;
7 : b=a;
```

This yields seven atoms, although at first glance it may appear as though there are eight. This is not the case because items #3 and #4 are a metaloop, and are thus treated as a single item. Paragen was applied to this problem using a population for 1000 for 50 generations, and generated the following *best-of-run* individual:

There are just five atom transformations applied, with two sets of metaloop genes. Two sets were chosen as that is the number of meta-loops in the original program. In this heavily data dependent case, all dependencies are preserved, with only a single pair of atoms being executed in parallel. However, a huge speed up is achieved by virtue of the transformation of the loops. The original program took 405 time-steps to execute, and now takes just six.

Notice for the first loop, the **Replace** transformation creates a parallel loop, while the **Skip** transformation is essentially a blank transformation, and doesn't cause any further modification. With the second metaloop, the loops will be fused if possible. As there are a

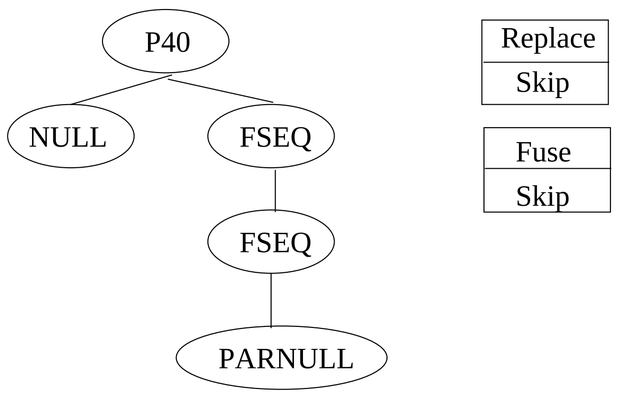

Figure 2.10
Best of run individual that produces the parallel code below. Notice how there is a set of linear genes for each metaloop.

number of ways in which this can be done, Paragen calculates the most useful, and applies that. This set of transformations is again terminated by a **Skip**, as no further modification is required.

The resulting parallel program is as follows:

```
0 : a=1;
1 : par-for (int i=0;i<100;i++)
      b[i]=a;
2 : a=10;
3 : par-for (int i=0;i<100;i++)
      {
      par-for(int j=0;j<2;j++)
        {
        d[i*j]=a;
        }
      d[i]=a;
      }
5 : a=100;
    par-begin
6 : f=a;
7 : b=a;
    par-end
```

As Paragen does not take communication into consideration however, it is unlikely that the program will remain at just six time steps. Another consideration is the number of pro-

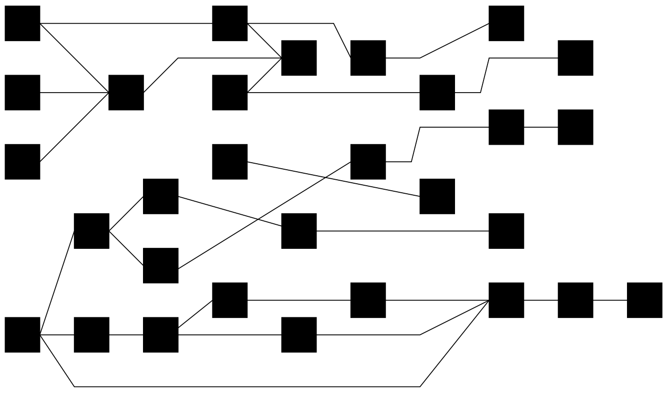

Figure 2.11
A parallel program viewed as a graph. The nodes represent instructions and the edges communication. Unlike many graphs, crossing edges do not cause any diffi culties.

cessors available. At its optimal rate, the program is executing 300 instructions in parallel, this means that when it is running on actual hardware, it will run *up to* 300 instructions at the same time, depending on the number of processors available.

2.4.5 Scheduling

One abstraction made at the start of the work was the assumption that all atoms require one time step, which makes it easier for Paragen to schedule atoms relative to one another. However, Paragen does not take into account atoms which require longer execution times, nor does it concern itself with communication costs.

As in the best spirit of computer science, the question of communication is catered for by a separate process. The code produced by Paragen assumes infinite processors, and so must be modified to run on specific machines. This is achieved by drawing a graph of the parallel program, and modifying the layout to reflect the number of processors involved in the runtime execution. Graph layout algorithms such as the Coffman-Graham[Coffman et al, 1972], Branch and Cut Approach[Junger et al, 1997] or Sugiuama[Sugiyama, 1981] can be used to reduce the number of crossings and adjacent, inter-column edges, which represent inter-processor communication. Figure 2.11 shows the sample output of a Paragen run, each node representing an instruction and each edge communication. Essentially, Paragen discovers *when* all instructions should be executed, while not making any comment as to *where* they should go. This must be dictated by the target architecture, taking into consideration both the number of processors and the speed of interprocess communication.

By using simple graph partitioning algorithms[Gerasoulis, 1990] we divide the instructions into tasks, which can then be mapped onto the parallel processors. Ideally, there will be much communication within these tasks, and inter-process communication will be reduced by ensuring that inter-dependent tasks reside on the same processor.

2.5 Conclusion

We have described an automatic software re-engineering tool which uses a modified version of Genetic Programming to evolve sequences of transformations which, when applied to a parallel program, generate a functionally equivalent parallel program.

GP is suitable for a task such as this because when one attempts to understand the logic of a program and then to convert that logic to an equivalent parallel form, the complexity grows enormously as the program increases in size. However, as GP applies its transformations without any understanding of the logic of the program which it is modifying, it scales far more graciously than other methods.

Moreover, while human programmers may have the advantage of being able to take a more holistic view of a program, GP is not only able to spot and exploit any patterns in the code, but also to take advantage of its bottom up approach. By this we mean it can move statements around, swap the location of loops and even join previously transformed loops together, in ways that human programmers are unlikely to think of.

Due to the arbitrary way in which GP cuts up and divides the code, and in particular, the loops, one can make no guarantee as to the readability of the modified code. In general, though, people are no more concerned about this than they are about being able to interpret the object code files from a compiler, so it is not a problem. There may be cases, however, where the end user of the code will want to be able to read it, e.g. if the programmer wished to add some extra, hardware-specific optimizations to the code. If this is strictly necessary, we can direct the loop transformations to insert comments where ever a loop is modified, thus providing an automatically documented piece of code.

Bibliography

Gerasoulis, A. (1990),"Dominant Sequence clustering Heuristic Algorithm for Multiprocessors", Tech Report Rutgers University, NJ.

Burns, A. (1988), *Programming Occam 2*, Prentice-Hall.

Braunl, T. (1993), *Automatic Parallelization and Vectorization, Parallel Programming : An Introduction*, Prentice Hall.

Coffman et al (1972), "Optimal Scheduling for two processor systems", Acta Informatica, 1, 200-213.

Davis, L.V. (1991), *Handbook of Genetic Algorithms*, Van Nostrand Reinhold.

Geist, G (1993), "PVM 3 Beyond Network Computing," in *Lecture Notes in Computer Science 734, 2nd Int. Conf. of the Austrian Center for Parallel Computation* pp. 194-203, Gmunden, Austria : Springer Verlag.

Gelerneter, D. (1985), "Parallel Programming in Linda", Technical Report 359, Yale University Department of Computer Science

Junger, M. and Mutzel, P.(1997) "2-layer straightline crossing minimisation : Performance of exact and heuristic algorithms" in *Journal of Graph Algorithms and Applications*, 1, 11-25.

Lewis, T. (1992), *Introduction to Parallel Computing*, Prentice Hall.

Piercom Ltd. http://www.piercom.ie

Peyton-Jones, S. (1992), *Functional Programming Languages*, Prentice-Hall.

Ryan, C. and Walsh, P. (1997), "The Evolution of Provable Parallel Programs". In *Genetic Programming 1997*, J. Koza et al (Eds.) pp295-302, Stanford, CA, USA : MIT Press.

Ryan, C. and Walsh, P. (1996), "Paragen: A novel technique of the Autoparallelization of Sequential Programs using GP". In *Genetic Programming 1996*, J. Koza et al (Eds.) pp197-204, Stanford, CA, USA : MIT Press.

Ryan, C. and Walsh, P. (1995): "Automatic conversion of programs from serial to parallel using genetic programming", in *Proceedings of Parallel Computing*, V Neagoe et al(Eds.), Gent, Belgium:Springer-Verlag.

Sugiyama, K. et al. (1981), "Methods for visual understanding of hierarchical system structures," IEEE Transactions on Systems, Man and Cybernetics, 11(2).