Two Evolutionary Representations for Automatic Parallelization

Kenneth P. Williams
Dept. of Computer Science,
University of Reading,
Whiteknights, Reading, RG6 6AY, UK
K.P.Williams@reading.ac.uk
Tel. +44 118 9875123, Ext. 7645

Shirley A. Williams
Dept. of Computer Science,
University of Reading,
Whiteknights, Reading, RG6 6AY, UK
Shirley.Williams@reading.ac.uk
Tel. +44 118 9875123, Ext. 8613

Abstract

In this paper we describe use of the REVOLVER system, a test-bed for experimenting with combinations of evolutionary representations and algorithms for automatic parallelization. Results show evidence of adaptation of auto-parallelization strategy by the evolutionary algorithms (EAs) tested, thereby suggesting that EAs are more capable of finding *enabling transformations* than current parallelizing compilers.

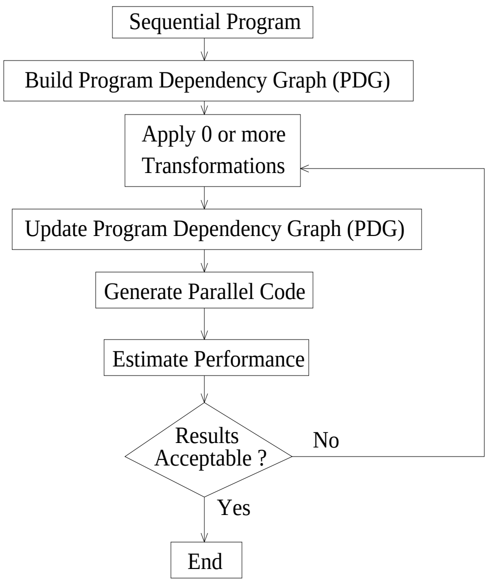

Figure 1: Typical Automatic Parallelization Process

1 Automatic Parallelization

Automatic parallelization is the automatic translation of sequential code into parallel form. This is a difficult problem since the parallel code must also perform efficiently on the target machine (with costly loss of performance if this is not the case) - hence it is an optimization as well as translation problem. The traditional approach has been to concentrate on restructuring computationally intensive loops using dependency analysis to determine which loops can be parallelized - and then using linear programming techniques during the code-generation phase to spread the workload evenly across the processors available. This approach (see Fig. 1) may be performed fully automatically (by a parallelizing compiler) or interactively (by a programmer guided by a parallelization tool). This analytical approach has only proved successful with highly-patterned, regular (i.e. a small, restricted-set of) programs.

One of the main difficulties encountered is identifying *enabling transformations* [1](i.e. transformations which by themselves have little, no, or possibly even a degrading effect of program performance but which enable other transformations (or sequences of transformations) to be made which *do* produce significant improvements in program performance. Conceptually, such transformations represent non-linearities in the search-space of semantically equivalent parallel programs. These transformations can be easily missed by a parallelizing compiler or by a programmer using an interactive tool.

One of the strongest motivations for particularly using evolution is that a lower-bound can be put on the quality of the solution produced by the evolutionary algorithm (EA). This can be achieved by the use of

[1] In this paper the terms *transformation* and *compiler optimization* are used interchangeably.

heuristic information and the incorporation of existing heuristics into the evolutionary algorithm which be done in such a way as to guarantee that the worst solution found will be at least as good as that found by existing techniques, *and probably better*. Extensive research into automatic parallelization over the years has lead to the development of a large body of heuristic and analytical techniques which may be included into such an evolutionary algorithm.

In section 2.1 we describe the Gene-Transformation (GT) representation and operators. In 2.2 we describe the Gene-Statement (GS) representation and operators. Section 3 describes the fitness function used with both representations. Experiments and test data are described in section 4 and the evolutionary algorithms (EAs) used in 4.1. A brief overview of the major results are presented in 5 with related research presented in 6. The paper finishes with a section of Conclusions.

2 Evolutionary Representations

It is well known that the choice of representation of a problem can often determine the success or failure of tackling the problem. In this section we present two representations of automatic parallelization (the *Gene-Transformation* representation, and the *Gene-Statement* representation) both of which are suitable for manipulation by evolutionary algorithms.

2.1 Gene-Transformation (GT) Representation

The first representation of the automatic parallelization problem developed (as presented in [8]) is called the 'gene-transformation' representation. As the name suggests, in this representation a single gene represents a single optimising transformation to be applied. Hence a whole chromosome (i.e. a sequence of genes) represents an ordered sequence of transformations to be applied. In our model, each transformation may be abbreviated by a three letter acronym, e.g.

LFU	Loop Fusion.
LSP	Loop Splitting.
LRV	Loop Reversal.
LIC	Loop Interchange.
...	etc ....

Currently the Revolver transformation catalogue only consists of loop transformations but other classes of transformations may be added and encoded in a similar style.

Our model also requires some means of referring to individual loops within a program. Simply, in a program containing n loops for example, we assign a number to each loop (from zero upwards) in lexicographic ordering. Hence all loops in the program will be assigned a unique number in the range $[\mathbf{0..}n\text{-}1])$. All loops are renumbered $[\mathbf{0..}n\prime-1])$ after application of each transformation (where $n\prime$ is the new number of loops in the program). All loop transformation functions currently take at least one parameter - the number of the loop they are to be applied to. For example:

LFU(6)	*"Fuse together loop 6 and any immediately following or else any immediately " preceding loop in the program."*
LIC(3)	*"Apply loop-interchanging to loop 3 in the program."*
LSP(7)	*"Apply loop-splitting to loop 7."*

Using the model we have defined so far we thus have a means of describing variable-length sequences of transformations (applied, in order from left-to-right), e.g.:

[*LSP(3), LRV(5), LFU(7)*]

This chromosome consists of three genes which encode the sequence:

"Apply a loop-splitting transformation to loop 3 in the program followed by loop-reversal to loop 5 and then finally loop-fusion to loop 7."

Given the representation thus far a number of issues now arise which need to be addressed, namely:

1. What to do if we try to apply a transformation which cannot be legally applied to the loop specified (i.e. the transformation 'fails')?
2. We need clearer definitions of how a number of transformations are to be applied (notably loop-fusion and loop-interchanging).
3. What evolutionary operators does the representation naturally give rise to?

These questions are answered next.

2.1.1 Decoders

It may happen in the course of restructuring that a particular transformation cannot be successfully applied. The failure to apply a transformation may arise for two reasons (i) it is *semantically* invalid (i.e. application of the transformation would violate the loop-dependency relations), or (ii) it is *syntactically* invalid (i.e. the loop is not suitable for the transformation, such as trying to apply loop-interchange to a single loop rather than a loop-nest, or trying to apply loop-fusion to a loop which is not immediately adjacent to another). The strategy of a traditional parallelizing compiler towards such a failure is to simply ignore the failed transformation and continue, however in the context of an EA how a failed transformation is handled has a significant effect on the movement of the population of transformation sequences as they progress across the search-space. As such this issue deserves closer scrutiny. Failure to apply a transformation may be handled in one of three ways:

- *delete-and-continue* (`D-A-C`): if a transformation cannot be applied, simply delete the gene (i.e. the transformation and loop-number) from the chromosome and continue to apply the following transformations specified in the sequence.
- *delete-and-stop* (`D-A-S`): if a transformation cannot be applied, delete the gene from the chromosome and delete the rest of the sequence.
- *repair* (`REPAIR`): if a transformation cannot be applied, replace the gene with a randomly generated new gene. If this new gene cannot be applied then repeatedly generate and test new genes until a legal one is created - (essentially the approach is to 'repair' the chromosome so it contains a valid sequence of transformations.

The process of resolving these conflicts is called *decoding*. The current version of REVOLVER implements all three of these *decoders*. A *decoding-strategy* therefore simply tells the compiler what to do when it comes across a transformation that cannot be applied. Which decoding strategy is employed may be specified by the user at run-time as a compiler-switch (the default value is *delete-and-continue*).

2.1.2 Transformation Definitions

The implementation of many compiler transformations arises naturally from their definition. For example, for a loop to be in 'normal form' means that its index variable is initialised to 1, and its increment-size is 1. Any counted loop (such as the F77 `DO` loop) can be transformed into normal-form simply by adjusting the loop-bounds and any array-acccess expressions to preserve loop behaviour. Hence, the implementation of this transformation (loop-normalization) is clear and unambiguous. However, some transformations may be implemented in more than one way. A detailed discussion of this issue is beyond the scope of this paper,

$$\begin{aligned}
\textit{Gene} &= \textit{Transformation \& Loop number.}\\
\textit{Chromosome} &= \textit{Sequence of Transformations.}\\
\textit{Mutation Operators} &= \textit{Inflict a small change on}\\
&\quad\ \textit{a sequence of Transformations.}\\
\textit{Crossover Operators} &= \textit{Recombine two sequences}\\
&\quad\ \textit{of Transformations.}
\end{aligned}$$

Figure 2: Summary of Gene-Transformation Representation

however we note that the strict implementation of a transformation will have a significant effect on the performance of any parallelization tool. Here we briefly describe only how loop-fusion and loop-interchanging are implemented in REVOLVER .

In REVOLVER the loop-fusion (LFU) transformation is implemented as follows: first, attempt to fuse the specified loop with an immediately following loop, if this is not possible then attempt fusion with an immediately preceding loop, if this is not possible then the transformation fails.

The loop-interchange (LIC) transformation is implemented in the following manner: first, if the specified loop is not the outermost loop of a perfect loop-nest then the transformation fails. Otherwise, if the depth of the loop-nest is two, then simply interchange the two loops, else (i.e. depth > 2) generate some random permutation of the loops and attempt interchange accordingly.

2.1.3 Operators and Algorithms

Using the 'gene-transformation' representation, what operators naturally arise, and what evolutionary algorithms work best with this representation?

The notion of mutation usually involves making small random changes to the chromosome (with the aim of improving the 'fitness' of the chromosome). Hence, three mutation operators immediately come to mind - (i) mutate the transformation specified by the gene, (ii) mutate the loop specified by the gene, and (iii) mutate the transformation *and* the loop specified by the gene (i.e. the whole gene). These three mutation operators, (all implemented in REVOLVER) currently replace mutated information with bounded-random values - more purposeful replacement of information using heuristic information may be implemented in future work. The notion of crossover usually involves *re-*

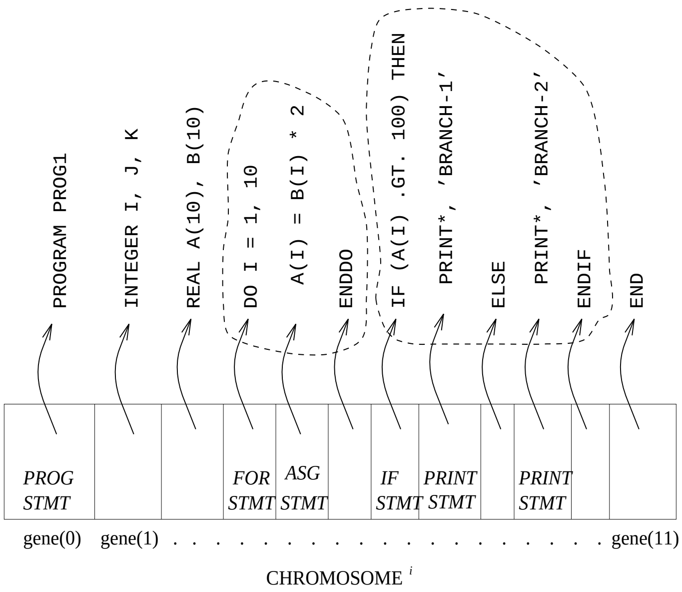

Figure 3: Gene-Statement Representation of a Fortran-77 Program (turn Figure right-sideways 90°)

combining two chromosomes in some meaningful way so as to produce two new offspring. Using the 'gene-transformation' representation two factors had to be considered in the design of our new crossover operators: (i) we are working with variable-length chromosomes, and (ii) the type of problem we are tackling is a scheduling-type of problem where the preservation of good sub-sequences of transformations is a desirable property of any operators we create. The popular one and two-point crossover operators adapted to work with variable length chromosomes would be useful in preserving good sub-sequences of transformations. Such operators have been implemented in Revolver and are referred to as *VLX-1* (Variable-Length Xover, 1-point) and *VLX-2* (Variable-Length Xover, 2-point) respectively.

A uniform crossover operator would have a disruptive effect on any potentially useful sub-sequences of transformations, yet some disruption is necessary for the search to make progress. Hence, a compromise was reached and a new operator created. Operator *VLX-3* protects developing sub-sequences of transformations by only applying uniform crossover to the final two-thirds (i.e. centre and right-hand two-thirds) of the two parent chromosomes. The first transformations to be applied (i.e the left-hand third of the chromosomes) are protected and not subjected to the uniform crossover operation. Having applied the mutation and crossover operators, we then need to select

$$
\begin{aligned}
\textit{Gene} &= \textit{Statement} \\
\textit{Chromosome} &= \textit{Program} \\
\textit{Mutation Operator} &= \textit{A type of Transformation.} \\
\textit{Mutation Operation} &= \textit{An application of} \\
&\ \textit{a Transformation.} \\
\textit{Crossover Operator} &= \textit{Not used.}
\end{aligned}
$$

Figure 4: Summary of Gene-Statement Representation

which members are to survive into the new population. Simple *binary tournament selection* (where repeatedly the fittest of 2 randomly chosen members of the current population is placed into the next generation, until the new population is full) is the only selection operator currently used in Revolver with the 'gene-transformation' representation. Most evolutionary algorithms can be made to work with this representation - some however, are more suitable than others. The current algorithms used in Revolver with this representation are random-mutation hill-climbing, simulated annealing, genetic algorithms, and self-adaptive genetic algorithms. A summary of the GT-representation is in Fig. 2.

2.2 Gene-Statement (GS) Representation

In the gene-statement representation (see Fig. 3), as the name suggests one gene represents one statement in the program. Hence, a sequence of genes represents a sequence of statements (i.e. a program). Each program statement has a type (e.g. a DO_STATEMENT, an IF_STATEMENT, an ASSIGNMENT_STATEMENT, etc). Transformations are seen as mutation operators which are applied to statements. In this context there is no 'meaningful' need for a crossover operator. Chromosome lengths may increase/decrease as mutations are applied - application of a loop-fusion/mutation (for example) will reduce the number of statements in the program (i.e. the number of genes in the chromosome), application of a loop-splitting/mutation will increase the number of statements in the program (i.e. the number of genes in the chromosome). A summary of the GS-representation is in Fig. 4 (*cf.* Fig. 2). Since transformations are now mutation-operators, each needs to be assigned a probability. (e.g.

```
pLRV = 0.02;
pLSP = 0.05;
```

```
begin
  t := 0
  initPopulation P ( t )
  evaluate P ( t )
  while ¬terminate do
      t := t + 1
      P' := selectParents( P )
      LSPmutation( P' )
      LRVmutation( P' )
      LICmutation( P' )
      . . . (allow other mutations)
      LFUmutation( P' )
      evaluate( P' )
      P := P'
  end
  printResults()
end
```

Figure 5: Gene-Statement Evolution Strategy

```
pLFU = 0.15;
pLIC = 0.10;
............
```

The mutation operators can now be applied probabilistically as they sweep across the genes of each chromosome in the population (Figs. 5 and 6). A meaningful order in which the operators may be applied then needs to be worked out. The optimizing phases of a typical parallelizing compiler (as described in [10]) gives us a useful guide. (Essentially, loop-splitting should initially be applied so as to allow more potential for restructuring to take place, in the final phases loop-fusion should be applied so as to increase the workload of slave processes). It is noteworthy that the GS-representation has the appearance of a population-based, iterative, probabilistic version of a standard parallelizing compiler (i.e. it is essentially an extension of current parallelizing compiler design).

3 Fitness Function

Once a sequence of transformations has been applied to a member of the population (i.e. the sequential program has been restructured) parallel code is generated. Several important features of this code are analysed (e.g. communication patterns, operations count, process granularity) and an estimated execution-time is returned. This estimate forms the 'fitness' value of the sequence of transformations which produced the code (in the GT representation), and the fitness value of the actual code produced (in the GS representation). The techniques used in computing this estimation are the

```
for p := 1 to POPSIZE
do
  stmt := p → firstStatement()
  while (stmt) do
    if (stmt → type() == FOR − STMT)
then
      x = rand()
      if (x < pLSP) then
        following = stmt → followingStmt()
        success = ApplyLSP(stmt)
        if success then
          stmt = following
        else
          stmt = stmt → followingStmt()
        fi
      fi
    fi
  end
end
```

Figure 6: Example Mutation-Transformation Algorithm

same as those used in the VFCS parallelizing compiler system and are fully described in [2].

4 Experiments

A test suite of 5 programs (4 publicly available F77 benchmark programs plus 1 specially constructed example) were used to test REVOLVER . The programs were `LFK-18` and `ADI` (Livermore Loops Fortran Kernels), `EFLUX` (Perfect Benchmarks) and `M44` generic code [2] plus `TEST-1`. The REVOLVER system was implemented for automatic parallelization of Fortran-77 programs onto a message-passing Meiko CS-1 network of 12 transputers and consisted of a profiler, dependency analyzer, loop-transformation catalogue, static performance-estimator, and code-generator - as well as the EA codes.

4.1 Evolutionary Algorithms (EAs)

In the following experiments we use six evolutionary algorithms. Each algorithm uses one of the two representations described and, if using the GT-representation only, will also be using one of the three decoding strategies described in section 2.1.1. Furthermore, six genetic operators (described in section 2.1.3) will also be available to the EAs, although not all can be used with the various algorithm/representation combinations we experiment with - none of the crossover operators can be used with

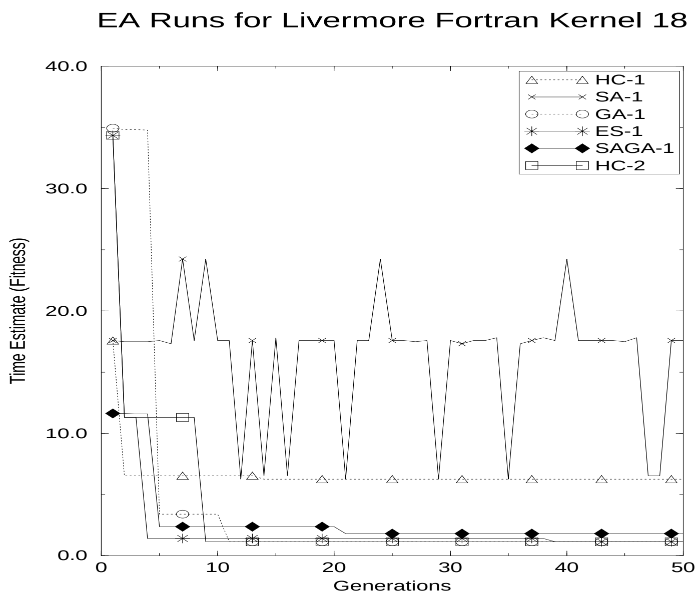

Figure 7: EA Runs on Livermore Fortran Kernel-18

the hill-climbing or simulated-annealing algorithms, and are not used with the GS-representation, for example. Six Evolutionary Algorithm/Representation combinations are defined.

HC-1 Steepest-ascent hill-climber using GT representation.

HC-2 Steepest-ascent hill-climber using GS representation.

SA-1 Standard simulated-annealer using GT representation.

ES-1 Evolution-strategy using GS representation.

GA-1 Genetic-algorithm using GT representation.

SAGA-1 Self-adaptive genetic-algorithm using GT representation.

Further parameters are defined, such as population size (`POPSIZE` = 5) and number of generations (`MAXGENS` = 50). In the results presented each run was for 50 generations (experiments showed that with the Revolver system most improvements occurred within the first 50 generations). After some experimentation operator probabilities were set: crossover operators 0.5 and mutation operators 0.2 except in `SAGA-1` where these were starting-values and the operator probabilities were changed as the run progressed. Where the GT representation was used, the population of transformation-sequences were randomly initialised to chromosomes of 5 transformations in length.

5 Results

Each of the 6 EAs defined in 4.1 was run 10 times on each of the 5 test F77 programs. In each run the parameters were exactly the same (as above) the only difference being the random number generator was seeded differently each time. Space restrictions preclude a complete exposition of results (see [9]) however, by far the clearest result found overall was that EAs using the GS representation consistently produced more significantly efficient code than EAs using GT (emphasising the importance of representation over algorithm). An example of the progress of the EAs can be seen in Fig. 7. This also confirms our idea of the terrain of the solution-space being searched by the EAs as being uneven and irregular.

Table 1: t-Test comparisons of EAs using fittest values returned from 10 runs on each of the 5 test programs.

HYPOTHESIS	*CONFIDENCE VALUE*
GA-1 out-performed HC-1	38%
SAGA-1 out-performed HC-1	30%
GA-1 out-performed SAGA-1	9%
ES-1 out-performed GA-1	99%
ES-1 out-performed SAGA-1	99%
ES-1 out-performed HC-2	53%
HC-1 out-performed SA-1	8%
ES-1 out-performed HC-1	100%
ES-1 out-performed SA-1	100%
HC-2 out-performed SA-1	100%
GA-1 out-performed SA-1	44%
SAGA-1 out-performed SA-1	36%
HC-2 out-performed GA-1	93%
HC-2 out-performed SAGA-1	95%

5.1 Comparison of Algorithms and Representations

The results of the runs for each EA were tabulated and confidence values were computed to compare the effectiveness of each EA across all runs. These confidence values are summarised in Table 1. The dominance of GS-based algorithms is clear.

5.2 Comparison of Decoding Strategies

Decoding strategies are only necessary with the GT-representation. As before, we are able to calculate confidence values of all comparable results. These values are presented in Table 2. Use of the `REPAIR` strategy is the clear winner. This result is interesting because

Table 2: t-Test Comparisons of Decoding Strategies.

HYPOTHESIS	*CONFIDENCE VALUE*
REPAIR out-performed D-A-C	88%
REPAIR out-performed D-A-S	98%
D-A-C out-performed D-A-S	50%

the reasons for the `REPAIR` strategy performing significantly better than the other two are not clear. It might be expected that the `D-A-S` strategy may not perform so well since the approach may prevent the population from sampling a large proportion of the search space. However this is not the case for `D-A-C` which was expected to perform roughly equal with `REPAIR`.

5.3 Auto-Parallelization Strategies

Generating parallel code for a loop introduces time overheads of communication (sending data), message/process start-up times, and synchronisation (between processes waiting for data) - these overheads must be offset against the gain of executing the loop workload in parallel. Most parallelizing compilers perform extensive memory and communication optimizations to minimize these overheads that have yet to be implemented in Revolver . In all the results presented so far the code-generator has worked by generating parallel code for all loops which contain no cross-iteration dependencies regardless of whether the loop workload made it worthwhile or not - and whether the loop was in normal form (section 2.1.2) or not (i.e. pre-normalisation of loops prior to code generation is *disabled*). The alternative is to normalise all loops before attempting to generate parallel code (i.e. pre-normalisation of loops prior to code generation is *enabled*). This is significant since in our transformation catalogue we have one transformation which can take a loop out of normal form, namely Loop-Reversal (LRV). By application of LRV the EA can influence whether or not a loop *may* be parallelized. The idea is that by not forcibly normalising loops before code-generation we are giving the EA the option of preventing parallelization of code by applying Loop-Reversal and hence taking a loop out of normal form.

The point is it became possible to compare the performance of EAs working under two sets of conditions (i) do not normalise all loops prior to code-generation (the default strategy), called *preNorm-disabled*, and (ii) normalise all loops prior to code-generation called *preNorm-enabled*.

Comparison of performance results shows that at no time did EAs using *preNorm-enabled* generate better code than EAs using *preNorm-disabled* (due to the introduction of expensive communications overheads and the result of the code-generator performing no optimizations). Most significantly, under the two sets of conditions two different auto-parallelization strategies emerged. Examination of the code produced shows that when pre-Normalisation was disabled - the EAs adopted the strategy of '*keeping loops sequential*' (by repeatedly applying loop-reversal) so as to avoid incurring message communications overheads that would slow the program down. When pre-Normalisation was enabled - the EAs adopted the strategy of '*making loops larger*' (by applying loop-fusion) so as to increase the size (the *granularity*) of the parallel processes created when loops are parallelized. In short, under two different sets of conditions the EAs were able to *adapt* their own strategy for automatic parallelization - *without any user intervention*.

6 Related Research

Research into automatic parallelization can be traced back to the vectorizing Fortran compilers of the late 1960's. More recent research has concentrated on parallelizing code for a wider range of languages (C/C++) and aiming at both shared-memory and message-passing MIMD architectures. Many of the compiler techniques developed over this time are described in [1]. Such a body of work also serves as an extensive source of heuristic techniques for incorporation into an EA.

The main evolutionary approach to automatic parallelization previously has been to use genetic programming techniques to restructure sequential code [7] into equivalent parallel form. This eliminates the need for costly dependency analysis but introduces the requirement for stringent checking to preserve program correctness.

The work presented in [8] was the first description known to the authors of using a genetic algorithm to automatically parallelize sequential code (using an early version of the GT representation). This idea was adapted for automatic parallelization onto a shared-memory multi-processor in [6] which reported a 21-25 % performance improvement against parallel code simply generated using the `Petit` /UTF parallelization libraries [4]. In [9] a comprehensive description of the Revolver system is presented introducing the GS and GT representations, operated on by various combinations of evolutionary algorithms and (some novel) operators. In [5] a hybrid genetic algorithm for task allocation (HGATA) of workloads to a multicomputer keeping the workload balanced and communication down to a minimum, is described. A novel approach utilising neural networks (using a technique called 'cellular encoding') is used to parallelize Pascal programs as described in Gruau *et al* [3].

7 Conclusions

The main findings of this research are (i) evidence of adaptation by the EAs at the auto-parallelization strategy level (i.e. a high level of abstraction) - this strategy was changed by the EAs in order to optimize the performance of the code it generated under the conditions in which it found itself. In our experiments this meant direct intervention with the code-generator - which caused the EA to change strategy (from '*keeping loops sequential*' to '*maximise the granularity of processes*'). It follows that an EA which is capable of adapting its own auto-parallelization strategy is significantly more likely to find enabling-transformations (than simply using traditional techniques) and hence produce better quality code - due to the non-linear effects of applying transformations. (ii) Demonstration of the clear superiority of the GS representation over GT (probably due to the strong interaction between the genes (epistasis) that GT suffers from).

Executing an EA will take longer, and use more computational power than existing parallelizing compilers. However the time requirement can be offset by the user receiving intermediate 'best solutions so far' found by the EA. In order to receive higher quality solutions the EA will have to execute for a longer period of time than traditional compilers - however, it is felt that most users will accept this because of the great importance attached to restructuring code to achieve the highest degree of performance possible and the expensive costs of failure to do so.

Acknowledgements

The authors thank Dave Corne (University of Reading, UK) and Paul Walsh (University College Cork, Ireland) for comments during this work. Part of this research was financed under grant 94701308 from the Engineering and Physical Sciences Research Council (EPSRC) of the UK.

References

[1] Bacon, D.F, Graham, S.L., and Sharp, O.J., *Compiler Transformations for High-Performance Computing*, ACM Computing Surveys, Vol 26, No. 4, pgs 345-420, Dec (1994).

[2] Fahringer, T., *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers*, Ph.D Thesis, University of Vienna, Austria, Sept., (1993).

[3] Gruau, F., Ratajszczak, J-Y., and Wiber, G., *A Neural Compiler*, Theoretical Computer Science, 1834, pgs 1-52, Elsevier, (1994).

[4] Kelly, W., Maslov, V., Pugh, W., Rosser, E.J., Shpeisman, T., and Wonnacott, D., `Petit` *Version 1.00 User Guide*, Omega Project, Dept. of Computer Science, University of Maryland, April (1996).

[5] Mansour, N., and Fox, G.C., *A Hybrid Genetic Algorithm for Task Allocation in Multicomputers*, in Proceedings of the 4th International Conference on Genetic Algorithms (ICGA-4), pgs 466-473, Morgan Kaufman Publishers, San Mateo, CA, (1991).

[6] Nisbet, A., *GAPS: Genetic Algorithm Optimised Parallelisation*, invited presentation - 7th Workshop on Compilers for Parallel Computing, Linkoeping, Sweden, June, (1998).

[7] Walsh, P., and Ryan, C., *Automatic Conversion of Programs from Serial to Parallel Using Genetic Programming - The Paragen System*, in Proceedings of PARCO-95 (Parallel Computing 1995), Universiteit Gent, Belgium, 19 - 22 September, (1995).

[8] Williams, K.P., and Williams, S.A., *Genetic Compilers: A New Technique for Automatic Parallelisation*, in "Parallel Programming Environments for High Performance Computing", Proceedings of the 2nd European School of Parallel Processing Environments (ESPPE-96), IMAG-INRIA, L'Alpe d'Huez, France, April 1-5, pgs 27-30, (1996).

[9] Williams, K.P., *Evolutionary Algorithms for Automatic Parallelization*, Ph.D Thesis, Department of Computer Science, University of Reading, Reading, UK, Dec. (1998). `http://www.cs.reading.ac.uk/cs/research/Publications/theses.html`

[10] Wolfe, M.J., *High Performance Compilers for Parallel Computing*, Addison-Wesley, (1996).