A Genetic Algorithm for Automatic Parallelization

Kenneth P. Williams, Shirley A. Williams

Abstract— **The limited ability of automatically parallelizing compilers to find implicit parallelism in sequential programs is a major barrier to the improved use of parallel computers. Optimizations for parallel architectures aim to maximize parallelism with transformations based on extensive control and data dependency analysis. However, current parallelizing compilers lack an organising framework that enables the direct calculation of the optimal sequence of loop transformations to be applied.**

The proposed solution is to use an evolutionary algorithm to find a sequence of loop transformations which, when applied to a sequential program, produce a parallelized version of the program that executes optimally (in a master/slave mode of parallelism) on any specified parallel architecture. The evaluation function of such an approach requires using static performance estimation techniques to determine the 'fitness' of the code produced. This indicates the creation of a hybrid genetic algorithm (GA) which incorporates control and data dependency analysis into the design of the new genetic operators.

We present the REVOLVER system which implements a genetic algorithm to automatically parallelize sequential Fortran-77 programs for a 16-node Meiko CS-1 message-passing architecture.

Keywords— **Genetic Algorithms, Automatic Parallelization, Static Performance Estimation.**

I. Introduction

WRiting good programs for parallel computers is a time-consuming and difficult task. To implement the program optimally the programmer needs to have detailed understanding of the underlying machine architecture. Features such as the memory hierarchy, interconnection topology, bandwidth size, processor architecture, task scheduling and processor allocation all have to be taken into account in the design and implementation of the program.

Manual translation of existing sequential programs into parallel form can be even more difficult. The programmer also needs a clear understanding of the functionality of the sequential program before being able to rewrite the program in a suitable parallel style. If the sequential implementation was poor then identifying areas of implicit parallelism can be difficult.

Interactive parallelizing tools are available but here the programmer additionally needs to know about parallelizing transformations, how to combine them, and the affects they have on program performance - all within the constraints imposed by the dependency analysis. In short, if the program is to be parallelised to achieve an acceptable level of performance then it needs to be customized for each target architecture it is to run on. This presents a major barrier to producing portable parallel software and hence increased use of parallel computing generally.

K.P.Williams and S.A.Williams are with the Department of Computer Science, University of Reading, UK. E-mail: {K.P.Williams,S.A.Williams}@reading.ac.uk .

Source to source program restructuring allows a single incarnation of a sequential program to be transformed at source code level so that it is more able to take advantage of the capabilities offered by any given target architecture.

In this paper we lay the foundations for an evolutionary approach to automatic parallelization. We specify how the problem of automatic parallelization can be encoded into a form suitable for manipulation by a genetic algorithm. We develop operators to transform programs and - importantly, indicate how problem-specific information can be incorporated to guide the evolutionary algorithm.

II. Model of Automatic Parallelization

Our strategy for automatically parallelizing sequential programs consists of four stages (see Fig.1):

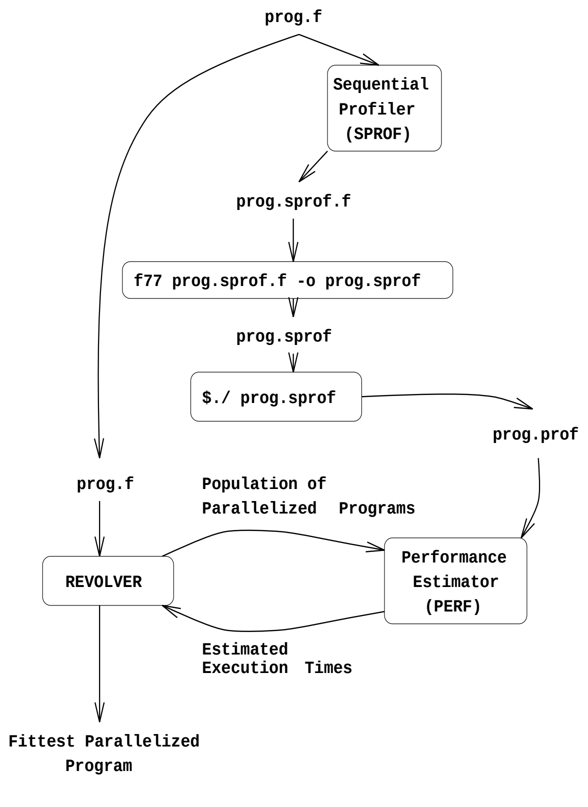
prog.f
Sequential Profiler (SPROF)
prog.sprof.f
f77 prog.sprof.f -o prog.sprof
prog.sprof
$./ prog.sprof
prog.prof
prog.f
Population of Parallelized Programs
REVOLVER
Performance Estimator (PERF)
Estimated Execution Times
Fittest Parallelized Program

Fig. 1. Automatic Parallelization Strategy

1. An instrumented version of the sequential program (`prog.f`) is created using the sequential profiling tool (SPROF). Instrumentation code is inserted to:

(a) Count the execution frequency of the programs' basic blocks.

(b) Compute the TRUE/FALSE ratios for each conditional statement (i.e. for each conditional statement, count how many times dufring the program run it evaluated to TRUE and to FALSE. Each conditional is then assigned a T/F ratio figure between 0.0 and 1.0 indicating how many times it eavluated to TRUE. The advantage of assigning a ratio is that the figure is scalable.

(c) Count the number of times each loop iterates. If the number of loop iterations varies, then the average number of iterations is simply taken.

(d) Insert timing code to record the amount of time the program spends in various sections of the code (e.g. loops, subroutines, etc).

The instrumented program (`prog.sprof.f`) is then compiled and executed, accumulating profile data into the file `prog.prof`. It should also be noted that the profiling run should aim to be a *typical* execution of the program. This profiling data is later used to estimate the performance of the parallel code generated.

2. Create a randomly initialised population of sequences of optimising and restructuring transformations (see §III-A). Each of these sequences is then applied in turn to the original sequential program tom produce a population of parallelized versions of the program. (Note: the transformations may only be applied within the constraints imposed by dependency analysis of the program (see §II-A).
3. Assign to each member of the population a fitness value. The 'fitness' of a parallel program is the estimated execution time (which we aim to minimize) as determined by the static performance estimator, PERF, (see §III-E).
4. Apply a number of genetic operations to the population of restructuring transformations to produce a new population of sequences of transformations. These are then applied to the original program (as in stage 2, above) to produce a new population of parallelized versions of the program.

Stages 3 and 4 are then repeated for a specified number of generations, or until the population converges onto one highly fit parallel program (or until some time-limit has been reached). The output of the GA is one equivalent version of the sequential program, which has been parallelized to some optimal degree. The sequence of transformations which were applied to produce this program may also be output.

A. Dependency Analysis

Dependence relations are used in the restructuring process to identify the reordering constraints placed upon the execution of statements. These constraints are required to preserve the behaviour of the program as it undergoes restructuring to ensure that any parallelized program produced is semantically equivalent to its sequential original.

The process of dependency analysis typically involves computing *DEF* and *USE* sets for program statements as detailed in [1]. Using this information, control and data dependencies between program statements can be computed. These constraints are encapsulated in a program abstraction called the *program dependence graph* (PDG), which must be initially calculated, and then updated after each transformation is applied.

A.1 Scalar Dependency Analysis

In simple terms, given the following sequence of statements;

```
S1 :    A = B + C
S2 :    D = A * 2
S3 :    E = A + 3
```

Statement S_1 must execute before statements S_2 and S_3 since the variable A is assigned a value in S_1 which is used in those statements. Also note that there is nothing to prevent statements $S2$ and S_3 from executing in parallel. Furthermore, statement S_1 could actually be eliminated by applying a dataflow transformation, which would result in the following code (which contains no constraints on parallelism).

```
S1 :    D = (B + C) * 2
S2 :    E = (B + C) + 3
```

A.2 Dependency Analysis for Arrays

So far we have only considered dependency analysis of individual memory locations. Dependency analysis of loops that iterate over arrays is the most fruitful area of analysis for exposing implicit parallelism. Loop iterations can be executed in parallel without explicit synchronisation if and only if there are no dependencies between instructions belonging to different iterations. If dependencies do exist then restructuring of the loop into a form amenable to parallelization may be attempted. In the following loop, it is clear from examining the loop index and the array subscript expressions that no dependencies exist between iterations and the loop may be executed in parallel;

```
S1 :    for i = 1 to 10000 do
S2 :        A[i] = B[i] / 5 * 9 + 32
S3 :    endfor
```

The following loop nest however,

```
S1 :    for i = 1 to 10000 do
S2 :        for j = 1 to 10000 do
S3 :            A[3*i+2*j, 2*j] = ........
S4 :            ........ = A[i-j+6, i+j]
S5 :        endfor
S6 :    endfor
```

requires extensive integer programming analysis techniques to determine that this loop nest has no dependencies and may also be executed in parallel. Note also, that the restructuring problem is further compounded in the cases of loops with variable loop bounds, non-linear subscript expressions, unstructured programmng constructs, and embedded subroutine calls and I/O.

A.3 Omega Test

The depndency analysis method used in Revolver is an integer programming technique called the Omega Test [8], [?]. This technique uses an extended form of Fourier-Motzkin elimination [5] to determine if solutions exist to the dependency equations reprsentative of the loop indices and array accesses.

III. The 'Revolver' System

The Restructuring EVOLVER system implements a GA with variable length chromosomes (GAVaCL [11]). Fortran-77 programs are first normalised [3] to remove any extraneous language features, all loops are then identified and numbered from 1..N, as are all functions/subroutines. Dependency analysis is then performed and the restructuring process begins. The system was implemented using the *Sage*$^{++}$ restructuring compiler libraries [15] and uses the *Petit* libraries [13] to perform dependency analysis.

A. Chromosomal Representation

An instance of a sequence of transformations to be applied is encoded by a chromosome as a string of genes, where each gene is a record of 4 integer fields, namely;

(Transformation-Number, Parameter-1, Parameter-2, Parameter-3)

The *Transformation-Number* filed is a unique integer representation of a transformation (as defined in the Transformation Catalogue), see Fig.2. The parameter fields are optional fields used to store any parameters associated with the tranasformation. Unused parameter fields are assigned zero. So for example, the single gene in Fig.2 can be read as "*apply loop-splitting to loop number 4 in the program*". The first 3 transformations of the genotype in Fig.2 can be read as "*fuse together loops 2 and 3, followed by splitting loop 4, then in-lining procedure 4 at call-site number 2*".

B. Conflict Resolution Strategies (CRSs)

In the process of applying a sequence of transformations, the restructurer may be asked to apply a transformation it is unable to do so. This may occur for 2 reasons:

1. Dependency analysis prevents application of the transformation since application would violate the associated dependency constraints and alter the behaviour of the program.
2. Due to the dynamic nature of the restructuring process, references to non-existant loops/subroutines may occur. This may happen for example when a transformation specifying (LSP,10,0,0) is encountered but due to previous transformations, loop 10 no longer exists.

These conflicts need to be resolved 'intelligently' - currently 2 methods are being investigated.

(a) Given a chromosome, continue applying transformations until the first invalid transformation is reached, then stop. The resultant code is then parallelized and evaluated as per normal.

```
/* Define Transformation Catalogue */

(LFU) = Loop_Fusion      =  1      /* Fuse two loops into one */
(LSP) = Loop_Splitting   =  2      /* Split a loop containing N statements
                                      into N loops. */

(IFC) = IF_Conversion    =  3      /* Make all IF-statements suitable for
                                      dependency analysis */

(LIC) = Loop_Interchange =  4      /* Interchange 2 perfectly nested loops */
   .              .
   .              .
   .              .


/* Define function prototypes and parameters */
int Loop_Fusion (int loop1, int loop2) ;
int Loop_Splitting (int loop1);
int IF_conversion () ;
int Loop_Interchange (int loop1, int loop2) ;
int Procedure_In_Lining (int procedure_no, int call_site) ;
```

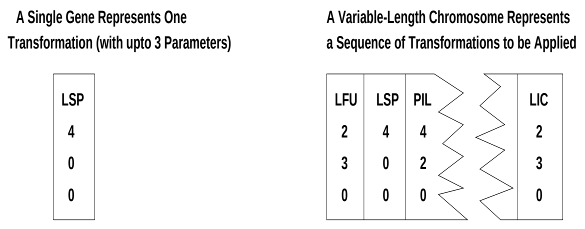
A Single Gene Represents One Transformation (with upto 3 Parameters)
A Variable-Length Chromosome Represents a Sequence of Transformations to be Applied
LSP 4 0 0
LFU 2 3 0
LSP 4 0 0
PIL 4 2 0
LIC 2 3 0

Fig. 2. Chromosome Representation of a Sequence of Transformations

(b) Given a chromosome, applying all legal transformations. If an invalid transformation is encountered simply delete it and continue along the chromosome.

The comparative impact of these 2 CRSs are still unclear and are the subject of ongoing investigation [19].

C. Genetic Operators

The current genetic operators arise naturally from the representation. Most notable, is that they must be capable of operating on chromosomes of variable lengths.

- Transformation Mutation : Mutate the value of the Transformation-Number in a gene, (e.g. BEFORE (LSP, 3,0,0) AFTER (LNO, 3,0,0)). This has the effect of changing the type of transformation applied at a point in the program.
- Parameter Mutation : Mutate the value of a parameter in a gene, (e.g. BEFORE (LNO 3,0,0) AFTER (LNO,4,0,0)). The loop-normalisation transformation (LNO) will now be applied to a different loop in the program.
- Gene-Mutation : Mutate the Transformation-Number and parameters within a gene, (e.g. BEFORE (LRV,3,0,0) AFTER (LFU,1,2,0).
- Unroll and Fuse : This operator directly encodes the known heuristic of applying the loop-unrolling transformation followed by a loop-fusion transformation to loops in close lexicographic proximity. It works by searching a chromosome for an instance of the loop-unrolling transforma-

tion (LUR) and then inserting a loop-fusion transformation immediately afterwards.

• 1-point and 2-point Crossover : These operators may prove useful since they maintain sub-sequences of transformations (which may be appropriate for our GA).

There is no lower bound on the length of a chromosome (except '1'), since only one transformation may be needed in order to produce optimal parallel code. An arbitrary upper-bound may currently be set by the user as an additional parameter of the GA.

Most importantly of all, the development of appropriate conflict resolution strategies and operators must take into account their effects as the populastion moves through the fitness landscape.

D. Master/Slave Parallelism

Once the code has been restructured it is parallelized for recompilation on the target architecture. The mode of parallelism used is the master/slave mode. One master process (the main program) spawns N slave processes for each parallelized segment of code. Each slave performs the same operations but works on different data which it receives from the master process. Upon completion each slave sends its results back to the master. The master receives the results from all N slaves, then resumes its execution.

This model is a simple yet efficient form of parallelism which involves a regular set of communication patterns. This property can be used to make static performance estimation easier and more accurate.

Imposing this model of parallelsim is also a limitation of the Revolver system, since not all problems are best restructured into the master/slave model. Other models of parallelism (e.g. pipelined, SPMD, 2D-grid, etc) could form an extra gene on the chromosome of restructuring transformations in future work.

E. Fitness Evaluation

Once the code has been restructured a performance estimation is calulated which returns a time-figure prediction of how long the parallelized program will take to execute. The estimator tool (PERF) has been constructed espcially for the Revolver system for analysing message-passing Fortran-77 programs, using a master/slave mode of parallelism and running on the Meiko CS-1. There are twpo main areas of code which are analyzed :

1. Communication Costs : Benchmark timings are performed to derive average time costs of typical communication patterns. A value of the message start-up time is also estimated. The parallelized code is then analyzed for (i) Number of messages sent, and (ii) sized of each message (in bytes). An accumulated time-cost value is then computed.
2. Operator Costs : Bench mark timings for the cost of typical binary operations (assignment, addition, subtraction, etc) are made using various combinations of data types (integer, real, complex, etc). The cost of executing each statement is then calculated and combined with information gathered during the profiling run (using SPROF) to gain a morre accurate figure (particularly with counting loop iterations, and TRUE/FALSE evaluations of conditional statements).

The two figures are added to produce a final estimate of how long the program will take to execute. This figure represents the 'fitness' of the parallel code produced.

IV. Related Work

Automatic vectorization was the earliest method of finding parallelism in sequential programs. The ILLIAC-IV, CEDAR, and Parafrase [12] projects at the University of Illinois, and the Parallel Fortran Converter (PFC) project [2] at Rice University, were the first projects to develop dependency analysis to automatically determine vector parallelism. Parafrase could be retargeted at vector, array-processor, and shared-memory multiprocessor architectures. Its features included automatic construction of data dependency graphs, construction of strongly connected components (π-block) graphs, and a large (over 100) program transformation catalogue to improve parallelism. Much of the foundation of automatic parallelization for message-passing machines was established with the SUPRENUM project [21] in Germany in the 1980's. A family of retargetable restructuring tools called KAP (Kuck and Associates Parallelizer), have been successfully developed for commercial use [9]. The SUIF (Stanford University Intermediate Form) compiler group [16] is an ongoing project where the source code has been made publicly available to encourage research into optimising and restructuring compiler techniques. Several schemes to avoid the "one transformation at a time" paradigm have been proposed. The unimodular transformations framework of Banerjee [5] has provided a particularly efficient means of parallelizing perfectly nested loops.

Balasundaram *et al.* [4] describe a static performance estimator used in conjunction with the distributed memory Fortran-D programming system to guide the programmers decisions when the parallelization tool is used in interactive mode. The method involves benchmarking the time-costs of a number of typical communications and operations on the target machine then applying cost-model analysis to produce a final estimate of the program's performance. Fahringer [6] describes an approach combining data accumulated during a profiled execution of the sequential program, with benchmarked costs of a range of operations, in order to produce an accurate, retargetable parallel program performance estimator for use as part of the Vienna Fortran Compilation System (VFCS) [17].

Evolutionary approaches to automatic parallelization so far have been few. Gruau describes a neural network approach to translate PASCAL programs into parallel form via an intermediate representation called 'cellular encoding'. In [18], Walsh and Ryan describe a system implementing a genetic programming approach for evolving OCCAM programs from sequential into parallel forms. A hybrid-GA for task allocation on multi-computers (HGATA) is presented in [10].

It should be emphasized here that all the approaches mentioned above, as well as our approach, aim at producing good sub-optimal solutions, and not necesarily the optimal, in a reasonable time.

V. Experiments

Not yet.

VI. Conclusions

The attractions of taking an evolutionary approach to automatic parallelization are many:

- Requires no user interaction or specialist knowledge.
- The restructuring GA may be parallelized.
- Any existing and future analysis techniques may be incorporated into the GA.
- The restructuring strategies of current parallelizing compilers may be incorporated into the GA to provide a lower-bound on the quality of the parallel code produced.
- GA may be executed off-line and/or as a background process, thus saving on expensive computational resources.
- The GA may be retargeted at any parallel architecture, provided an accurate performance estimator can be constructed [6].

One may further envisage other ways in which evolutionary techniques may be applied to the problem of automatic parallelization [19]. Like any search technique the performance of the GA can be improved by the inclusion of further heuristic information. Current research includes parallelizing the publicly available NAS (Numerical Aeronautics Simulation) programs [?].

Acknowledgments

The authors would like to acknowledge the useful comments of David Corne (University of Reading) and Andy Nisbet (Novel Computing Centre, University of Manchester, UK), in this work.

References

[1] A.V.Aho, R.Sethi and J.D.Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley (2nd Edition), 1986.

[2] Randy Allen & Ken Kennedy *Automatic Translation of Fortran Programs to Vector Form*, ACM Transactions on Programming Languages and Systems (ACM-TOPLAS), Vol 9, 4, pgs 491-542, Oct. 1987.

[3] David F. Bacon, Susan L. Graham & Oliver J. Sharp *Compiler Transformations for High-Performance Computing*, ACM Computing Surveys, Vol 26, No. 4, pgs 345-420, Dec. 1994.

[4] Vasanth Balasundaram, Geoffrey C. Fox, Ken Kennedy, & Ulrich Kremer *A Static Performance Estimator to Guide Data Partitioning Decisions*, in Proceedings of 3rd ACM SIG-PLAN Symp. ACM PPOPP, April 21-24, 1991.

[5] Utpal Banerjee *Loop Transformations for Restructuring Compilers, Volume 1 : The Foundations*, Kluwer Academic Publishers, 1993.

[6] Thomas Fahringer *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers*, PhD. Thesis, University of Vienna, Austria, Sept. 1993.

[7] Frederic Gruau, Jean-Yves Ratajszczak & Giles Wiber *A Neural Compiler*, Theoretical Computer Science, 1834, pgs 1-52, Elsevier, 1994.

[8] Wayne A. Kelly *Optimization within a Unified Transformation Framework*, Ph.D Thesis, Department of Computer Science, University of Maryland, (CS-TR-3725), Dec. 1996.

[9] R.H.Kuhn, B.Leasure, and S.M.Shah, , *The KAP Parallelizer for DEC Fortran and DEC C Programs*, Digital Technical Journal, Vol 6, 3, Summer 1994, pgs 57-70, 1994.

[10] Nashat Mansour & Geoffrey C. Fox *A Hybrid Genetic Algorithm for Task Allocation in Multicomputers*, in Proceedings of 4th International Conference on Genetic Algorithms (ICGA-4), Morgan Kaufman Publishers, San Mateo, CA, 1991.

[11] Zbigniew Michalewicz *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, New York, 3rd Edition, 1996.

[12] David A.Padua, David J.Kuck, Duncan H.Lawrie, *High Speed Multiprocessors and Compilation Techniques*, IEEE Transactions on Computing, Vol C-29, 9, pgs 763-776, Sept. 1980.

[13] Wayne A.Kelly, V.Maslov, W.Pugh, E.Rosser, T.Shpeisman, & D.Wonnacott *Petit Version 1.00 User Guide*, Omega Project, Dept. of Computer Science, University of Maryland, April 1996.

[14] William Pugh & David Wonnacott *Going Beyond Integer Programming with the Omega Test to Eliminate False Data Dependencies*, Tech. Report (CS-TR-3191), Department of Computer Science, University of Maryland, Dec. 1992.

[15] P.Beckman, J.Gotwals, S.Srinivas, D.Gannon & F.Bodin *Sage^{++} : A Class Library for Building Fortran and C^{++} Restructuring Tools*, 2nd Object-Oriented Numerics Conference, Oregon, USA, 1994.

[16] R.Wilson, R.French, C.Wilson, S.Amarasinghe, J.Anderson, S.Tjiang, S-W Liao, C-W Tseng, M.Hall, M.Lam, and J.L.Hennessey, *An Overview of the SUIF Compiler System*, Computer Systems Lab, Stanford University, 1993.

[17] S.Benkner, S.Andel, R,Blasko, P.Brezany, A.Celic, B.M.Chapman, M.Egg, T.Fahringer, J.Hulman, E.Kelc, E.Mehofer, H.Moritsch, M.Paul, K.Sanjari, V.Sipkova, B.Velkov, B.Wender, H.P.Zima *Vienna Fortran Compilation System : Version 1.2 - User's Guide*, Institute for Software Technology and Parallel Systems, University of Vienna, Austria, Feb. 1996.

[18] Paul Walsh & Conor Ryan, *Automatic Conversion of Programs from Serial to Parallel Using Genetic Programming - The PARAGEN System*, in Proceedings of Parallel Computing 1995 (ParCo-95), Universitëit Gent, Belgium, 19 - 22 September, 1995.

[19] K.P.Williams *Evolutionary Algorithms for Automatic Parallelization*, Ph.D Thesis, Department of Computer Science, University of Reading, UK, *expected*, Dec. 1997.

[20] Michael J. Wolfe *High Performance Compilers for Parallel Computing*, Addison-Wesley, 1996.

[21] Hans P.Zima & Barbara Chapman *Supercompilers for Parallel and Vector Computers*, ACM Press, New York, 1991.

Ken Williams will process your compuscript, adding "value" to your LaTeX file using a sophisticated text style for all to admire.

Shirley A. Williams will process your compuscript, adding "value" to your LaTeX file using a sophisticated text style for all to admire.