Parallel Functional Island Model Genetic Algorithms through Nested Algorithmic Skeletons

Greg Michaelson and Norman Scaife

Department of Computing and Electrical Engineering,
Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS
greg/norman@cee.hw.ac.uk

Abstract. Island model genetic algorithms(GAs) are based on independent GAs which evolve separately, and intermittently exchange genetic material. Such models may be expressed as nested higher-order functions and realised as the corresponding nested algorithmic skeletons. Here we consider the use of an island model GA for the Traveling Salesperson Problem(TSP) in the evaluation of our parallelising compiler for Standard ML. We also discuss how an island model GA may be used to solve the over-determined linear equations used in the compiler to predict parallel behaviour from sequential prototyping, and show that it is a very poor technique compared with Singular Value Decomposition.

1 Introduction

Genetic algorithms(GAs) [Gol89] are a class of evolutionary algorithm. They are based on the manipulation of sequences of encodings, termed genomes composed of genes. A population of such genomes may evolve optimally, relative to a fitness criteria, through:

- mutation, where random genes in random genomes are changed arbitrarily
- crossover, where random genes from random genomes are exchanged
- evaluation, where a fitness function is used to score each genome
- selection, where only the fittest genomes in a population are retained for subsequent evolution

GAs are widely used to try and solve problems where analytic techniques prove intractable, typically involving the traversal of very large domains.

In [MB99] we discussed the flat parallel realisation of a panmictic (i.e. single population) GA for the Traveling Salesperson Problem (TSP) from a Standard ML prototype, using a GA library based on that for Quadstone's RPL2 GA programming language [Sur93]. Essentially, mutation, crossover and fitness functions are repeatedly `map`ed over a population. To implement such GAs, we used our parallelising compiler for Standard ML [MSBK00], which automatically converts higher-order functions(HOFs) to algorithmic skeletons. Thus, `map`s are implemented as process farms.

Below, we discuss how the SML-based panmictic GA for the TSP may be extended naturally to an island model GA, and consider the behaviour of its parallel realisation through our compiler on the Fujitsu AP3000. We then present the profiler used in our compiler to predict parallel behaviour from sequential prototypes. Finally, we show how an island model GA might be used to solve the over-determined linear equations that result from profiling but that this is not an appropriate technique.

2 The panmictic GA for the TSP

In the TSP, a salesperson has to travel to a number of interconnected cities, taking the shorted possible route, visiting city only once and ending at the initial city. Figure 1 shows a map of interconnected cities. A map may be represented

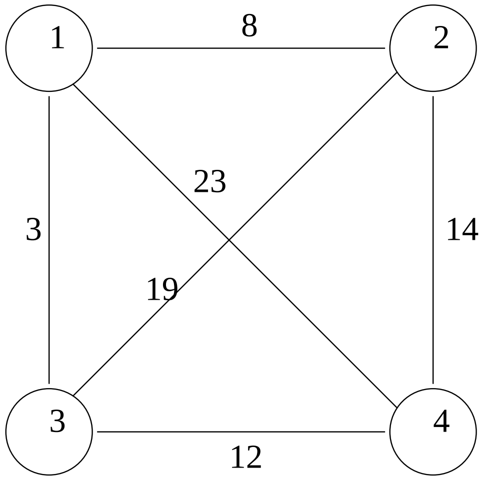

Fig. 1. Traveling Salesperson Problem map

as a list of pairs of cities and distances, for example:

```
val TSPmap = [(1,2,8),(1,3,3),(1,4,23),
              (2,3,19),(2,4,14),
              (3,4,12)]
```

A genome corresponding to a route may be represented as a list of cities, for example:

```
val route = [1,4,2,3]
```

Note that there is an implicit connection between the last and first city and that for N cities there are $N!$ possible routes. A population of routes may be represented by a list of route genomes, for example:

```
val routes = [[1,4,2,3],
              [2,3,2,1],
              [3,1,4,2]]
```

Note that the second route is invalid as city 2 is visited twice, at the expense of city 4.

The heart of the panmictic GA for this form of the TSP was:

```
 1  val cities = numbs (CITIES-1); (* generate city identifiers *)
 2  val country  = makemap cities; (* generate random map *)

 3  fun test () =
 4  let val (gstack(pgwH,nE)) = makeinit nPopsize [] nPopsize;
 5      val gsPop = gstack(map (fitness cities country) pgwH,nE);
 6      val gsPop' = RankRaw gsPop false;
 7      val gsPop'' = repeat 0 nextgeneration gsPop'
 8  in gsPop''
 9  end

10  and breed (gsPop,gsCache) =
11  let val gParentA =
12       SelectRawTournament gsPop false 2 SELRATE true;
13      val gParentB =
14       SelectRawTournament gsPop false 2 SELRATE true;
15      val gChild =
16       CrossNpt gParentA gParentB (CITIES-1) CROSSRATE
17  in (gsPop,Push gChild gsCache)
18 end

19 and nextgeneration gsPop =
20      let val (_,gstack(pgwH,nE)) =
21            ntimes nPopsize breed (gsPop,Empty())
22          val gsPop' =
23           gstack (map ((fitness cities country) o
24                        (Mutate CITIES MUTRATE)) pgwH,nE);
25          val gsPop'' = RankRaw gsPop' false;
26      in gsPop''
27      end;
```

In the RPL2 library, genomes are represented by their gene sequence and the associated fitness measure. Populations are represented by `gstacks` which consist of a stack of genomes and the number of genomes.

Here, line 4 creates the initial population `gsPop` and line 5 `maps` the fitness function over it. The `fitness` function checks that every city is represented once, penalising repeated cities, and then calculates the total distance for the route. A low fitness is best so line 6 sorts the population in increasing order of fitness.

Note that the best route is not known in advance so line 7 carries out evolution a fixed number of times, using `nextgeneration` at line 19.

Lines 20 and 21 apply crossover a fixed number of times to the initial population to form a new population, using `breed` at line 10. Lines 11 to 14 select two genomes for crossover at lines 15 and 16, to form a new genome which is added to a new population `gsCache` in line 17. Lines 23 and 24 use `map` to apply mutation to each genome in the new population and assess its fitness. Line 25 ranks the new population in increasing fitness order, to be returned at line 26.

In general, mutation and crossover are relatively low cost operations with the bulk of processing being expended on fitness calculations. Thus, in our parallelisation of this GA, parallelism is exploited in the `map` in lines 23 and 24. For further details, see [MB99].

3 An island model GA for the TSP

In an island model GA, a number of populations evolve separately but simultaneously in islands, to solve the same problem. Every once in a while, genomes may be exchanged between islands. Typically, a best genome will be selected for exchange, but also retained, and a worst genome will be replaced by an immigrant.

The use of island model GAs is motivated from nature, where it is observed that isolated populations may develop different specialisms depending on local conditions. Thus, Darwin[Dar10] observed that populations of finches (Figure 2) on different islands in the Galapagos group (Figure 3) had different adaptations

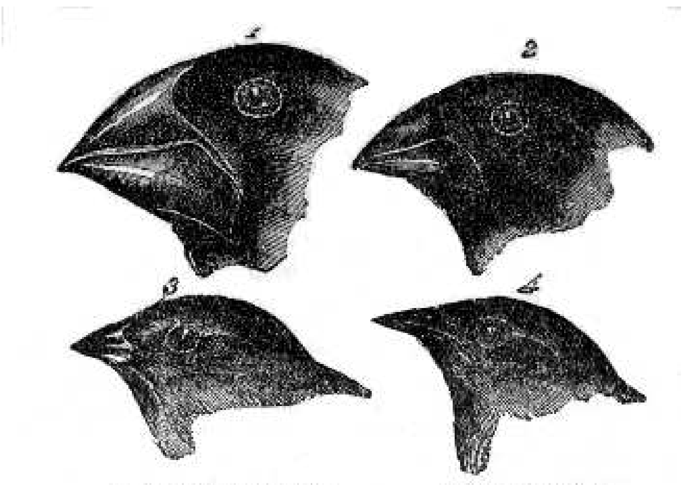

Fig. 2. Finches by C. Darwin

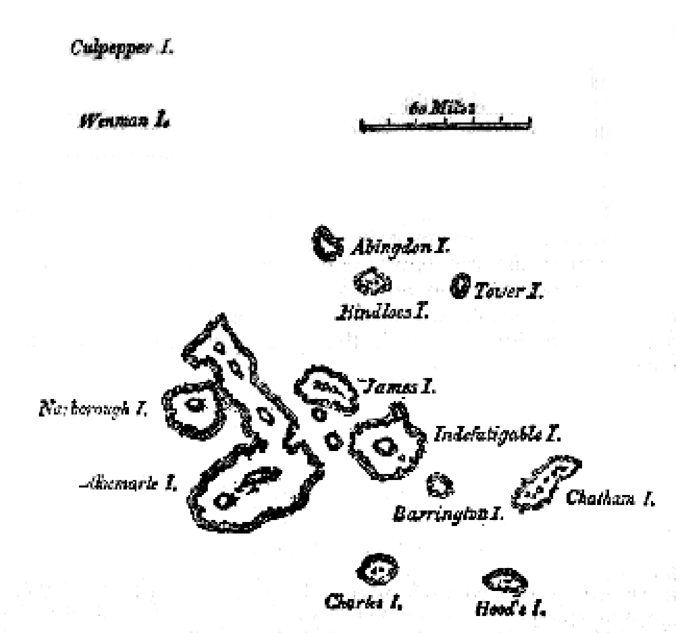

Fig. 3. Galapagos Islands by C. Darwin

to variations in vegetation and food. In nature, separate populations from an original species may evolve independently to the point where they become new species which can no longer inter-breed. In an island model GA, however, such speciation would have to be programmed explicitly.

In the TSP, the use of an island model might result in local specialisations for particular sub-routes, which cross island breeding might spread amongst other populations. In the above example, if one island evolved a local minima from city 3 to city 1, and another evolved a local minima from city 1 to city 4, then inter-breeding might result in a larger local minima from city 3 via city 1 to city 2.

A series of island populations may be represented by a list of routes, each a list, for example:

```
val islands = [[[1,4,2,3],[4,3,4,1],[2,3,2,1]],
               [[3,4,1,2],[2,2,2,1],[1,2,1,4]],
               [[1,3,1,3],[2,4,3,1],[2,3,1,4]]]
```

Independent island evolution may be effected by `maping` `nextgeneration` above over a sequence of islands. We choose to evolve all islands independently one step, find the best overall genome and substitute for the worst genome on each island:

```
fun run (b,p) =
     let val p' = map nextgeneration p;
```

```
3          val best = foldr findbest WORST p';
4       in (best,map (RNth best nIslandsize) p')
5       end
```

Here, `b` is the best genome so far from all islands and `p` is the islands, a list of lists of genomes. Line 2 maps `nextgeneration` over all islands, line 3 finds the fittest genome from all islands, relative to an initial arbitrary worst genome `WORST`, and line 4 replaces the Nth (i.e. the worst) on each island with the overall fittest.

Figure 4 shows times and speedups from the AP3000, on up to 9 processors, for one generation of finding the best route on maps of 20 and 40 cities, with 20 islands each of 50 genomes, with parallel `maps` for both `nextgeneration` and `run`. Here, speedup is the ratio of the time on N processors over the time on 1

	procs	time(secs)	speedup
20 cities	1	10.60	
	2	10.63	1.00
	3	7.69	1.39
	4	7.06	1.50
	5	6.56	1.62
	6	5.76	1.84
	7	6.74	1.57
	9	6.05	1.75
40 cities	1	120.41	
	2	125.86	0.96
	3	76.94	1.56
	4	71.45	1.69
	5	68.42	1.76
	6	68.20	1.77
	7	67.90	1.77
	9	66.39	1.81

Fig. 4. Times and speedups for parallel inner and outer `maps`

processor.

Note that the startup cost for multiple processors results in a worsening of times on 2 processors compared with 1 processor.

The times in the 40 city case are roughly 10 times those for the 20 city case. We noted above that the major cost in a GA is that of evaluating the fitness. Here, this involves searching an overall map, consisting of a list of pairs of city indices, to find distances between cities. For 20 cities the overall map is of size $20 * 19/2 = 190$ where as for 40 cities case the size is $40 * 39/2 = 780$, a factor of around 4. With 40 cities there will be, on average, twice as much searching as for 20 cities, giving an overall map factor of 8. Finally, with 40 cities there will be twice as much addition to find the total route distance. Thus, the parallel behaviour appears to scale well, as is reflected in the broadly comparable speedups.

However, in both cases, times are slow and speedups are very poor. To investigate this, Figure 5 shows times and speedups for 20 cities, with either the inner (`nextgeneration`) or the outer (`run`) `map` realised in parallel. Times are averaged from 25 generations.

	procs	time(secs)	speedup
parallel inner map	1	9.35	
	2	8.53	1.09
	3	13.63	0.68
	4	18.57	0.50
	5	13.34	0.70
	6	13.10	0.71
	7	17.35	0.53
	8	18.78	0.50
parallel outer map	1	4.82	
	2	4.77	1.01
	3	2.84	1.70
	4	2.12	2.27
	5	1.63	2.96
	6	1.61	2.99
	7	1.40	3.44
	8	1.02	4.73

Fig. 5. Times and speedups for parallel inner or outer `map`s with 20 cities

Parallelising the inner `map` is clearly ineffective. Parallelising the outer `map` is considerably more useful, giving modest speedups of around 50% of linear.

With more iterations on each island before selection, useful parallelism might be found on the inner `map` as well.

4 Profiling in the SML parallelising compiler

We have been developing a parallelising compiler for SML, where parallelism in HOFs is exploited through algorithmic skeletons. In our original conception, the usefulness of a HOF was to be determined through implementation independent sequential profiling, and parallel performance models for skeletons instantiated with architecture-specific parameters [MIK97].

Following Bratvold [Bra94], profiling is based on the structural operation semantics (SOS) for SML. Where Bratvold built an SOS-driven interpreter for his SML subset, we have modified the SML Kit II interpreter from DIKU [TBE+97] to capture counts for each rule for full core SML while a prototype is executed sequentially.

In the Skel-ML compiler, which was targeted at the Meiko Computing Surface, Bratvold used Busvine's low level costs from the INMOS transputer [Bus93] to calibrate his profiler. For our compiler, rather than capturing such low-level costs, we have chosen to find an overall cost for a test program on the target CPU and then relate it back to the SOS rule-count profile. Thus, for the ith test program, we wish to find coefficients C_i to solve:

$$R_{11}C_1 + R_{12}C_2 + ... + R_{1N}C_N = T_1$$
$$R_{21}C_1 + R_{22}C_2 + ... + R_{2N}C_N = T_2$$
$$...$$
$$R_{M1}C_1 + R_{M2}C_2 + ... + R_{MN}C_N = T_M$$

where:

M - number of tests
N - number of SOS rules
R_{ij} - count for jth SOS rule on ith test
C_j - coefficient for jth SOS rule
T_i - time for ith test on target architecture

Given a new program, function arguments to HOFs would be run on the sequential profiler to give new rule counts, say $NR_{k1}, ..., NR_{kN}$ for the kth HOF function argument. Predicted times for the arguments, say PT_k for the kth argument, would then be found by numerical solution of:

$$NR_{k1}C_1 + NR_{k2}C_2 + ... + NR_{kN}C_N = PT_k$$

using the coefficients found from the tests. Such predicted times would subsequently be used with performance models for the corresponding skeletons to estimate whether or not useful parallelism could be exploited from the HOFs.

To simplify equation solving, we ignore very small rule counts and those with known fixed cost, for example for unit `()` or for arithmetic operations.

When the number of tests becomes greater than the number of rules ($M > N$), this system of linear equations becomes over-determined. Currently, we use Singular Value Decomposition (SVD) to find the coefficients, but solutions have proved to be unstable as more tests are added.

5 Solving profile equations with an island model GA

As an alternative to SVD, we now consider the use of an island model GA to solve profile equations, with the potential for pleasingly self-referential parallel realisation through our compiler. At first sight, the use of a GA seems attractive, as one objective was to incrementally grow the accuracy of profiling prediction by adding profiles for new programs to the collection and then resolving the equations. It is plausible that having found a stable solution, a new stable solution for a new program could be found in a small number of iterations.

Here, each genome consists of a sequence of possible coefficients, say PC_1, PC_2, ..., PC_N. The fitness function evaluates the above linear equations with the genome values substituted for the required coefficients to find predicted times, say $PT_1, ..., PT_M$:

$$
\begin{aligned}
&R_{11}PC_1 + R_{12}PC_2...R_{1N}PC_N = PT_1\\
&R_{21}PC_1 + R_{22}PC_2...R_{2N}PC_N = PT_2\\
&...\\
&R_{M1}PC_1 + R_{M2}PC_2...R_{MN}PC_N = PT_M
\end{aligned}
$$

The overall fitness is then found from the sum of squares of differences between predicted and actual times:

$$\sum_{i=1}^{M}(PT_i - T_i)^2$$

The SML code required to realise the profile solving GA is very similar to that for the island model GA for the TSP above. The most important difference, apart from genomes consisting of `real` numbers, lies in the fitness function:

```
fun sumsqdiff ((v1:real),v2) t = (v1-v2)*(v1-v2)+t;

fun prod ((h1:real)::t1) (h2::t2) = h1*h2+prod t1 t2 |
    prod [] [] = 0.0 |
    prod _ _ = raise DIFF_LENGTH;

fun fitness' gl =
     foldr sumsqdiff 0.0 (zip times (map (prod gl) counts));
```

Here, `gl` is the genome whose fitness is sought, `times` is a list of observed test times, T_i, and `counts` is a list of lists of observed rule counts, R_{ij}. `prod` is `map`ed over the rule counts for all tests, to find the predicted times, $PT_1, ..., PT_M$, for the genome and each rule count. Finally, `sumsqdiff` is `fold`ed over predicted times `zip`ed with observed times, to find the overall fitness for the genome. The fitness calculations require more processing than for the TSP, and there might be additional parallelism in the `map` or `fold` in `fitness'`.

Sequential implementation of the profile solving GA, with 25 islands each of 100 genomes, for 134 test cases and 38 significant SOS rule counts, converges on a fitness of 6.36E 3 after 105 generations in 40 minutes. In contrast, SVD finds a solution with fitness 3.64E 6 in under a second.

Figure 6 shows parallel runtimes and speedups for each of the three parallel `map`s individually implemented in parallel, for one generation. The granularity of the outer `map` is very coarse and good speedups are achieved, with a speedup of over 9 for 16 processors. The inner `map` is too fine-grained for our methods and catastrophic slowdown is evident. This is so bad for larger numbers of processors that no results could be obtained. The intermediate `map` lies between these two cases in terms of performance.

	outer		middle		inner	
procs	time(s)	speedup	time (s)	speedup	time (s)	speedup
1	8.49		8.67		11.4	
3	4.88	1.74	5.69	1.52	95.0	0.12
4	3.26	2.60	4.47	1.94		
5	2.58	3.29	3.87	2.24	85.7	0.13
6	2.09	4.06	3.74	2.32	85.8	0.13
7	1.87	4.54	3.63	2.39	89.0	0.13
8	1.59	5.33	3.93	2.20		
9	1.50	5.66	3.15	2.75		
10	1.35	6.30	2.83	3.06		
11	1.27	6.67	3.02	2.87		
12	1.21	7.02	2.94	2.95		
13	1.08	7.87	2.96	2.92		
14	1.01	8.42	2.99	2.90		
15	0.99	8.53	3.01	2.88		
16	0.93	9.08	3.21	2.70		

Fig. 6. Times for single parallel outer, middle and inner `maps`

Figure 7 shows the performance of a parallel nested map implementation where the outer and middle `maps` are implemented in parallel. The inner `map` is implemented sequentially. Note that for 3 to 6 processors only the outer `map` is implemented in parallel but that for 7 to 16 processors both `maps` are implemented in parallel. There is a slight difference between a parallel `map` implemented sequentially at compile time and a decision by the run-time system to not allocate processors to a parallel-implemented `map`. In the former case the `map` is implemented purely sequentially but in the latter much of the initialisation code for the parallel map is implemented and the `map` is run sequentially *by the parallel* `map`. Note also that, due to the topological restriction that all nested skeletons at the same level must have the same number of allocated processors some of the results have redundant processors.

These results are less optimal than the singly-nested outer processor model but reasonable speedup of over 5 is achieved for 16 processors. In an application of this type, where no work is generated by intermediate nodes, our system will always have better performance with coarse-grain outer parallelism than fine-grained internal parallelism.

6 Conclusion

The main objective of this work was to test the behaviour of nested skeletons generated from HOFs by our parallelising compiler. While results from the island model TSP are in themselves modest, they confirm that our compiler can successfully generate nested algorithmic skeletons from HOFs in small scale SML programs.

procs	time(secs)	speedup	parallel `maps`	active processors
1	8.59		none	1
3	4.90	1.75	outer	3
4	3.30	2.61	outer	4
5	2.64	3.25	outer	5
6	2.15	4.00	outer	6
7	3.59	2.39	both	7
8	4.86	1.77	both	7
9	3.33	2.58	both	9
10	2.43	3.53	both	10
11	2.39	3.60	both	10
12	2.46	3.49	both	10
13	1.93	4.46	both	13
14	1.90	4.53	both	13
15	1.89	4.55	both	13
16	1.67	5.14	both	16

Fig. 7. Times for parallel outer and middle `maps`

The profile solver GA results for the triply-nested `map` implementation again show respectable speedup and confirm that for linear nested `maps` and coarse-grain MIMD implementation, the parallelism is most usefully found at the outer level. The method of solving complex multivariable linear equations using GAs has not been very successful in terms of performance since it does not compare favourably with more conventional linear algebra techniques. The speed of solving the equations is not, however, our greatest motivation for using this technique. The SVD method assumes that the fit function is *linear in its coefficients*. This turns out to be too restrictive for our data sets. We also need to apply constraints to the coefficients and, although there are methods for doing this in linear algebra, the SVD method would have to be replaced.

For future work, it would be very interesting to:

- compare island model GA behaviour from automatically parallelised SML programs with that from equivalent hand crafted parallel implementations
- analyse in considerably more detail the SOS rule-count behaviour of typical large SML programs rather than relatively simple artificial test cases

7 Acknowledgements

This work is supported by EPSRC grant GR/L42889. We would like to thank Quadstone Ltd for access to RPL2 documentation and advice on RPL2 use, and the Imperial College Fujitsu Parallel Computing Research Centre for use of their Fujitsu AP3000. We would also like to thank our colleague Paul Bristow, who developed the skeletons used in our compiler, and Hans-Wolfgang Loidl, Robert Pointon and Phil Trinder for comments on this paper.

References

[Bra94] T. Brativold. *Skeleton-based parallelisation of functional programs.* PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 1994.

[Bus93] D. Busvine. *Detecting parallel structures in functional programs.* PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 1993.

[Dar10] C. Darwin. *A Journal of Researches.* Ward Lock, 1910.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimisation and Machine Learning.* Addison-Wesley, 1989.

[MB99] G. Michaelson and P. Bristow. Parallel functional genetic algorithms in Standard ML from RPL2. In P. Trinder and G. Michaelson, editors, *Draft Proceedings of 1st Scottish Functional Programming Workshop*, Technical Report RM/99/9, pages 253–261. Heriot-Watt University, August 1999.

[MIK97] G. Michaelson, A. Ireland, and P. King. Towards a skeleton based parallelising compiler for SML. In C. Clack, T. Davie, and K. Hammond, editors, *Draft Proceedings of 9th International Workshop on Implementation of Functional Languages*, pages 539–546. University of St Andrews, September 1997.

[MSBK00] G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested algorithmic skeletons from higher order functions. *Parallel Algorithms and Applications, special issue on High Level Models and Languages for Parallel Processing*, September 2000.

[Sur93] P. Surry. RPL2 Functional Specification. Technical Report EPCC-PAP-RPL2-FS 1.0, University of Edinburgh/British Gas, 1993.

[TBE+97] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Hojfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with Regions in the ML Kit. Technical Report 97/12, Department of Computer Science, University of Copenhagen, 1997.