Parallel Functional Genetic Algorithms in Standard ML from RPL2

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

August 25, 1999

Abstract

RPL2 is an imperative genetic algorithm language, whose data-parallel constructs may be readily expressed as standard higher-order functions. Here we consider the conversion of RPL2 programs to Standard ML, to test our skeleton based parallelising compiler. Results from running a simple GA to solve a Travelling Salesperson Problem on a Fujitsu AP3000 are presented.

1 Introduction

Genetic algorithms(GAs) [1] are based on concepts from evolutionary biology and are increasingly used to solve problems which require the exploration of large search spaces. A *genome* encodes a potential solution to a problem as a linear sequence of symbols. The relative merits of such solutions are characterised by a *fitness function* which effectively interprets each genome and returns a *fitness value*. In a typical GA, a genome pool is initialised with random genomes. In each round of evolution, genomes are selected randomly for *cross over*, where symbols from one genome are exchanged with those from another. Each genome may also be subject to *mutation*, where symbols are changed arbitrarily. The fitness of each genome is then assessed using the fitness function. Finally, some fitness criteria is used to select and reproduce promising genomes to enter the next round of evolution. This continues until a maximally fit genome evolves, and is deemed successful if that individual is also optimally fit. The whole process may be summarised in pseudo-code as:

```
initialise genome pool
FOREACH genome in pool DO
 assess fitness
```

```
WHILE haven't found optimally fit genome DO
 crossover genomes with some probability
 FOREACH genome in pool DO
  mutate genome with some probability
  assess fitness
 select and reproduce fittest genomes with some probability
```

Most GAs follow this basic pattern. The key decisions in GA use are the choice of:

- the encoding of solutions as symbol sequences
- the fitness function
- the termination criterion.

For further details, see Goldberg[1].

2 RPL2

RPL2 is a Reproductive Plan Language for implementing genetic algorithms in parallel. It is based around a simple flat imperative language with data-parallel iterators for manipulating genome pools. RPL2's power derives from the extensive library of GA operations: thus, it might be considered as a GA coordination language. RPL2 was developed originally at the Edinburgh Parallel Computing Centre in a joint project with British Gas [2] investigating pipe-network optimisation problems. It is now maintained and distributed by Quadstone Ltd.

RPL2 provides:

- integer, real, boolean and string base types.
- assignable variables of the base types.
- the `genome` type for representing genome instances. While concrete representations are under programmer control, all genomes contain the symbol sequence and a raw fitness measure. A set of base `genome` operations is provided.
- the `gstack` type to represent a LIFO stack of `genomes`, with a set of base `gstack` operations.
- sequential bounded and unbounded iteration statements.
- conditional statements.
- data-parallel iteration over `gstacks`.
- structured `genome` populations.

- data-parallel iteration over structured populations.

The RPL2A libraries include:

- the standard instruction set, with generic operations for selection and replacement of `genomes` in `gstacks`, and for structuring `gstacks` by raw or scaled fitness criteria.
- the binary representation, for representing genomes as bit sequence.
- the variable cardinality integer representation, for representing genomes as integer sequences

The representation libraries provide representation specific functions for manipulating `genomes`.

To illustrate RPL2, consider the following example, modified from [2], for evaluating De Jong's F2 function, a standard test for GA systems[1]. This treats a genome as containing two binary fractions which are used to find $x1$ and $x2$ in the range $-2.048 <= x1, x2 <= 2.048$ by linear interpolation. The genome's fitness is then evaluated using the $F2$ function:

$$f_2(x_1, x_2) = 100(x_1^2 - x_2^2)^2 + (1 - x_1^2)^2$$

```
plan(PanmicticExample)
use StdInst, Binary(64);
int iCounter, nGenerations, i, nPopsize;
genome gNew,gChild,gParentA,gParentB;
gstack gsPop,gsCache;
Randomize(0)
for iCounter := 1 to nPopsize
 gNew := RandomGenome();
 Push(gNew,gsPop);
endfor
forall gNew in gsPop
 EvalDeJongF2(gNew);
endforall
for i := 1 to nGenerations
 Empty(gsCache);
 for iCounter := 1 to nPopsize
  gParentA := SelectRawTournament(gsPop, bMaxIsBest, 2, 0.5 True);
  gParentB := SelectRawTournament(gsPop, bMaxIsBest, 2, 0.5 True);
  gChild := CrossNpt(gParentA,gParentB,1,0.8);
  Push(gsCache,gChild);
  endfor
  forall gChild in gsCache
          Mutate(gchild,0.01);
          EvalDeJongF2(gChild);
```

```
25   endforall
26   Swap(gsPop, gsCache);
27   RankRaw(gsPop, bMaxIsBest);
28   StatsPrint (i,1,gsPop,"binex.out");
29 endfor
30 endplan
```

The standard instruction set and binary libraries are loaded (line 2). All `genomes` will now implicitly represent 64 bit binary genomes. Various variables are declared (lines 3 to 5) and the random number generator is initialised (line 6).

The `gstack gNew` is initialised to `nPopsize` random genomes (lines 7 to 10). Each genome's initial raw fitness is then set using the De Jong F2 operation (lines 11 to 13). Note that this is a parallel iteration.

Lines 14 to 29 are the main loop. Each time, the auxiliary `gstack gsCache` is emptied (line 15).

Next, repeated selection is made of two genomes, `gParentA` and `gParentB`, for crossover(lines 17 to 20). In a raw tournament, genomes are selected with equi-probability. The `2` indicates that the tournament is between two genomes and the `0.5` indicates the probability of the best in the tournament being selected. The genome `gChild` is then formed by swapping halves of each parent. The `1` in the call to `CrossNpt` indicates that there is one point where a genome may be split into segments for exchange, with segments taken alternatively from each parent. The `0.8` is the probability that a crossover takes place at all.

Then, each genome in `gsCache` is mutated, with probability `0.1` and has it's fitness assessed (lines 22 to 25). Again note the parallel iteration.

Finally, the new stack of genomes replaces the old stack, the new genomes are ranked in order and a report is printed (lines 26 to 28).

3 From RPL2 to SML

We are developing a skeleton based parallelising compiler from SML to C with MPI[3]. Potential parallelism is identified at sites of higher-order function use and realised through the instantiation of algorithmic skeletons[4]. The compiler is written in SML. The front end is based on the ML Kit. After analysis and annotation, SML abstract syntax trees are translated to Objective CAML. The Objective CAML compiler is then used to generate native machine code, linking in skeletons written in C with MPI. When the project started, we sought realistic applications with which to test the compiler: Quadstone suggested RPL2 as a promising source.

The RPL2 standard instruction set and binary libraries were re-written in SML, using list structures to represent `genomes` and `gstacks`. As with RPL2, Marsaglia and Zaman's universal random number generator is used[5].

The De Jong F2 example was then hand translated to SML and tested sequentially, to debug the libraries and gain experience of translation. Sequential iterations were realised as recursion using, `ntimes` below, and parallel iteration through `forall` as `map`:

```
val GLENGTH = 64;

fun makeinit 0 l nE = gstack(l,nE) |
    makeinit n l nE = makeinit (n-1) (RandomGenome GLENGTH::l) nE;

fun ntimes 0 f gs = gs |
    ntimes n f gs = ntimes (n-1) f (f gs);

val nGenerations = 100;
val nPopsize = 50;

fun test() =
let val _ = Randomize();
    val (gstack(pgwH,nE)) = makeinit nPopsize [] nPopsize;
    val gsPop = gstack(map EvalDeJongF2 pgwH,nE);
    val gsPop' = RankRaw gsPop false;
    val gsPop'' = ntimes nGenerations loop gsPop'
in gsPop''
end

and next (gsPop,gsCache) =
let val gParentA = SelectRawTournament gsPop false 2 0.8 true;
    val gParentB = SelectRawTournament gsPop false 2 0.8 true;
    val gChild = CrossNpt gParentA gParentB 1 0.8
in (gsPop,Push gChild gsCache)
end

and loop gsPop =
     let val (_,gstack(pgwH,nE)) = ntimes nPopsize next (gsPop,Empty())
         val gsPop' = gstack(map (EvalDeJongF2 o (Mutate 0.01)) pgwH,nE);
         val gsPop'' = RankRaw gsPop' false
     in gsPop''
     end;
```

We next decided to investigate the Travelling Salesperson Problem to test the compiler.

4 The Travelling Salesperson Problem

The Travelling Salesperson Problem(TSP) is a classic NP-Complete problem for which solutions may be found through GAs. Essentially, there are N interconnected cities and a shortest route between them is required, visiting each city once and returning to the first city. The problem has complexity $O(N!)$: there are N possible first cities, $N-1$ possible second cities, $N-2$ possible third cities and so on, giving $N*N-1*N-2... = N!$ possible routes.

A route may be encoded as a sequence of N city identifiers, with an implicit connection between the last and first. The fitness value is the length of the route so the fitness function must:

- check that a route does not contain duplicate cities; if it does then a maximally poor fitness may be returned
- return the sum of the distances between each city and the next in the sequence, including that from the last to the first

Routes may then be ranked in ascending order of route length, and those with the shortest routes are good candidates for further evolution. Given that the most optimal solution to a TSP cannot be determined without evaluating all possible solutions, the GA may be stopped either after a fixed number of evolutionary cycles regardless of the best route's fitness or when the best route has not changed after a fixed number of cycles.

The TSP is a well suited for solution by GA as in any set of cities there are optimal sub-routes: changes to these will decrease the fitness of genomes that encode them. Thus, in the long term, optimal sub-routes will tend to propagate through the genome population.

5 TSP implementation

For the TSP, a cut down version of the variable cardinality integer library was implemented, so that a route could be represented by a list of integer identifiers for each city in the sequence. A set of interconnected cities was represented by a list of pairs of city identifiers and inter-city distances.

The fitness function uses a list of unique city identifiers to check that a route contains each city, giving a penalty for each missing city. It then looks up successive pairs in the route and totals the distances between them to give the overall fitness:

Rather than write RPL2 for translation to SML, the De Jong F2 SML was modified:

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

fun test () =
let val (gstack(pgwH,nE)) = makeinit nPopsize [] nPopsize;
    val gsPop = gstack(map (fitness cities country) pgwH,nE);
    val gsPop' = RankRaw gsPop false;
    val gsPop'' = repeat 0 loop gsPop'
in gsPop''
end

and next (gsPop,gsCache) =
```

```
let val gParentA = SelectRawTournament gsPop false 2 SELRATE true;
    val gParentB = SelectRawTournament gsPop false 2 SELRATE true;
    val gChild = CrossNpt gParentA gParentB (CITIES-1) CROSSRATE
in (gsPop,Push gChild gsCache)
end

and loop gsPop =
     let val (_,gstack(pgwH,nE)) = ntimes nPopsize next (gsPop,Empty())
         val gsPop' = gstack (map ((fitness cities country) o
                                (Mutate CITIES MUTRATE)) pgwH,nE);
         val gsPop'' = RankRaw gsPop' false;
     in gsPop''
     end;
```

Note that `repeat` in `test` is like `ntimes` above in repeatedly applying `loop` to the current population `gsPop'` for a specified number of cycles. However it differs from `ntimes` in also printing progress information.

Note that `CITIES-1` crossover points are used in `next` where `CITIES` is the number of cities.

6 Parallel evaluation

The above program was compiled through our system, with a parallel map realised as a processor farm replacing the `map` in `loop`. Each processor has a local copy of the universal random number generator, initialised with the processor number as seed.

The program was run on a Fujitsu AP3000 with a population of 500 genomes, mutation probability of 0.01 (`MUTRATE`), crossover probability of 0.02 (`CROSSRATE`) and selection probability of 0.7 (`SELRATE`), varying the number of cities and processors. Each run was for 50 generations. Times in seconds are:

Procs	Cities								
	5	10	15	20	25	50	100	125	150
1	23	52	158	221	689	5223	34028	65164	106128
2	35	85	161	382	674	3100	39394	51005	170472
4	26	48	109	246	51	1855	13248	15333	32276
8	27	51	77	159	222	985	6650	13052	26211

and speedups (time for 1 processor/time for N processors) are:

Procs	Cities								
	5	10	15	20	25	50	100	125	150
1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2	0.7	0.6	1.0	0.6	1.0	1.7	0.9	1.3	0.6
4	0.8	1.1	1.4	0.9	13.5	2.9	2.6	4.2	3.3
8	0.9	1.0	2.0	1.4	3.1	5.3	5.1	4.8	4.0

In general, GAs are highly non-deterministic, depending utterly on random effects. This makes interpreting timings of runs with fitness based termination criteria very difficult as further runs with different random number seeds might produce wildly different times.

However, the GAs here terminate after a fixed number of cycles and are regular data parallel:

- Crossover and mutation both involve scanning and copying entire genomes so they contribute a fixed cost linear in the genome size.
- The fitness function contributes fixed costs linear in the genome size for finding the distance between pairs of cities and for summing the distances between cities.

Here, we can observe that:

- Increasing the number of processors has more impact as the number of cities increases, as might be expected given that communication costs are only linear in the number of cities.
- There are several isolated anomalies: for example 25 cities on 4 processors gives super-linear speedup. These are probably artifacts of the particular test run and require further investigation.

7 Conclusions

These are preliminary results: each run above was only carried out once. These should now be repeated and times averaged, to provide a more convincing picture of the parallel behaviour.

The GAs should be run with a maximal fitness termination criterion rather than for a fixed number of cycles, to clarify search space pruning behaviour.

More investigation is needed of garbage collection behaviour.

Overall, times are poor and speedups low. Nonetheless, the RPL2 based GAs in SML have proved very useful for debugging, testing and demonstrating our parallelising compiler.

A parser for RPL2 has been written in SML and work continues on automatic translation. A future goal is the exploitation of parallelism in the manipulation of structured genome populations.

8 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 Norman Scaife, who developed the compiler front end and integrated the entire compiler.

References

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

[2] P. D. Surry. RPL2 Functional Specification. Technical Report EPCC-PAP-RPL2-FS 1.0, Edinburgh Parallel Computing Centre, 1993.

[3] N. Scaife, P. Bristow, G. Michaelson, and P. King. Engineering a Parallel Complier for Standard ML. In K. Hammond, T. Davie, and C. Clack, editors, *Proceedings of 10th International Workshop on the Implementation of Functional Languages*, pages 213–225. University College London, 1998.

[4] M. Cole. *Algorithmic Skeletons: Structured Managemnet of Parallel Computation.* Pitman, 1989.

[5] G. Marsaglia and A. Zaman. Towards a Universal Random Number Generator. *Statistics and Probability Letters*, 8:35–39, 1990.