A Massively Parallel GP Engine in VLSI

Sven E Eklund
Computer Science Department
Dalarna University, SWEDEN
sven.eklund@ieee.org

Abstract - In this paper we propose the implementation of a massively parallel GP model in hardware in order to speed up the genetic algorithm. This fine-grained diffusion architecture consists of a large amount of independent processing nodes that evolve a large number of small, overlapping subpopulations. Every node has an embedded CPU that executes a linear machine code GP representation at a rate of up to 20,000 generations per second.

I. BACKGROUND

Genetic Algorithms (GA) and Genetic Programming (GP) are groups of stochastic search algorithms which were discovered during the 1960's, inspired by evolutionary biology. Over the past decades GA and GP have proven to work well on a variety of problems with little a-priori information about the search space [11]. However, in order for them to solve hard, human-competitive problems, like those suggested in [12], they require vast amounts of computer power, sometimes involving more than $10^{15\text{-}17}$ operations.

II. PARALLEL GA

It is a well-known fact that the genetic algorithm is inherently parallel, a fact that could be used to speed up the calculations of GP. The basic algorithm by Holland [9] is very parallel, but also has a frequent need for communication and is based on centralized control, which is not desirable in a parallel implementation.

An *efficient* architecture for GP should of course be optimized for the calculations and communication involved in the algorithm. However, it has also to be *flexible* enough to work efficiently with a variety of applications, which have different function sets. Also, the architecture should be *scalable* so that larger and harder problems can be addressed with more computing hardware.

By distributing independent parts of the genetic algorithm to several processing elements which work in parallel, it is possible to speedup the calculations. Traditionally, the parallel models have been categorized by the method by which the population is handled. The choice between a global and a distributed population is basically a decision on selection pressure, since smaller populations result in faster (sometimes premature) convergence. However, the choice also has a major effect on the communication need of the algorithm.

Bethke [5] made one of the first investigations of parallel implementations of GA in 1976. He described a global population with a partial exchange of individuals in successive generations. In 1981 Grefenstette described four different parallel implementations of GA, with both distributed and global populations [8].

One of the first real implementations of parallel GA was made by Tanese in 1987. She conducted studies of different topologies and migration rates on a distributed population model on a 64 processor N-CUBE system. In some experiments she reported super-linear speedup compared to sequential GA [16].

II.I. The Farming Model

With a global population the algorithm has direct access to all the individuals in the population, either by a global memory or by some type of communication topology, which connects several distributed memories. This parallel model is often referred to as the farmer-model or the master-slave-model [6]. A central unit, a farmer or master, controls the selection of individuals from the global population and is assisted by workers or slaves that perform the evaluation of the individuals.

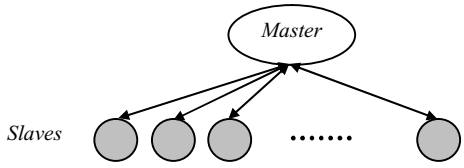

Figure 1 – The Farming model

This model has been reported to scale badly when the number of processing elements grow, due to the communication overhead of the algorithm [1], [2]. This is however heavily dependent on the ratio between communication time and computation time.

By dividing the population into more independent subpopulations, two alternative parallel models can be identified. Based on the size and number of subpopulations, they are referred to as coarse-grained or fine-grained distributed population models. When dealing with very large populations, which are common in hard, human-competitive problems, these models are better suited since their overall communication capacity scale better with growing population size.

0-7803-7282-4/02/$10.00 ©2002 IEEE

II.II. The Island Model

The coarse-grained, distributed population model, also known as the island model, consists of a number of subpopulations or "demes" that evolve rather independently of each other. With some migration frequency they exchange individuals between each other over a communication topology.

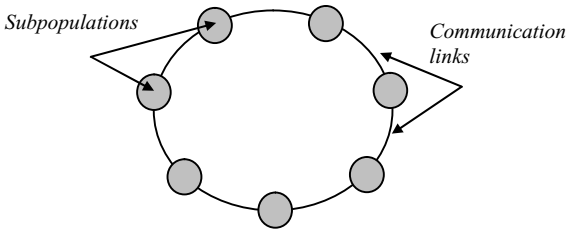

Figure 2 – The island model with seven demes in a ring topology

The island model is a very popular parallel model, mainly because it is very easy to implement on a local network with standard workstations (cluster). A major drawback of the island model is that it modifies the basic genetic algorithm and introduces new parameters, for instance the migration policy and the network topology. Today, there exists little or no theory on how to adjust those parameters [7]. Also, a system based on the island model is physically quite large, a fact that exclude many applications.

II.III. The Diffusion Model

The fine-grained distributed population model, often referred to as the diffusion model, cellular GA or massively parallel GA, distributes its individuals evenly over a topology of processing elements. It can be interpreted as a global population laid out on a structure of processing elements, where the spatial distribution of individuals defines the subpopulations. The subpopulations overlap so that every processing node belongs to several subpopulations, which makes the communication implicit and continuous and enables fit individuals to "diffuse" throughout the population in contrast to the explicit migration of the island model. Selection and genetic operations are only performed within these local neighborhoods [4].

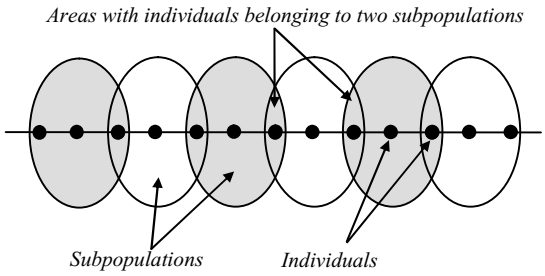

Figure 3 – The diffusion model with a linear topology

The diffusion model is well suited for VLSI implementation since the nodes are simple, regular and mainly use local communication. Since every node has its own local communication links over the selected topology (1D, 2D, hypercube, etc), the communication bandwidth of the system can be made to scale nicely with a growing number of nodes. Further, the nodes operate synchronously in a SIMD-like manner and have small, distributed memories, which also make the diffusion model suitable for implementation in VLSI.

III. IMPLEMENTATION

We believe that the massively parallel diffusion model is a promising model to implement *efficiently*, *scaleable* and *flexible* in hardware. A hardware implementation could also be made compact enough to be portable and possible to integrate, for instance, in mobile applications. In [15] several other advantages of the diffusion model are concluded, most notably the absence of a migration parameter and a potentially higher parallelism than in other parallel models.

Implemented in hardware the algorithm can execute very fast and will benefit directly from Moore's law. Machine code representation of the individuals, being one of the fastest ways to evaluate fitness [3], will further increase the performance making the implementation *efficient*.

Integrating a CPU in the diffusion node to evaluate the GP representation makes the system *scale* with a growing number of nodes. Since additional communication links also are introduced when the number of nodes grow, the system can balance communication with computation.

Finally, the use of machine code representation, an integrated CPU, and implementing the system in an FPGA will make the system highly *flexible*. Adapting it to different applications is simply done by changing the CPU's instruction set and recompiling the VHDL code with the appropriate fitness calculations.

III.I. Implementation

Lacking the explicit migration parameter of the island model, the diffusion model still has a number of parameters that need to be determined in order to optimize performance. In our first implementation of the diffusion model, we followed some of the recommendations made by Baluja [4].

The communication between nodes in the system is based on the X-net topology, a two-dimensional torus grid with extra diagonal links. The neighborhood consists of the center node and the eight nearest neighbors (Moore neighborhood). See figure 4.

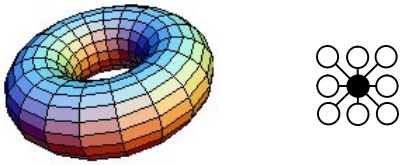

Figure 4 – Torus topology and the Moore neighbourhood

Please note that since all neighborhoods overlap in two dimensions, the black center node to the right in figure 4 will also belong to eight other adjacent neighborhoods.

Every node consist of four major parts, a CPU, a memory, a control block and a simple router as shown in figure 5. All nodes work in a synchronous SIMD-mode.

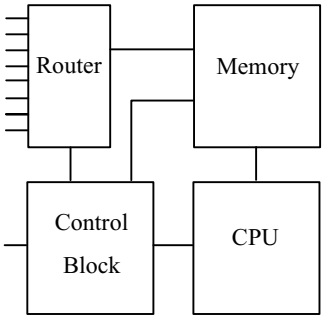

Figure 5 – The four blocks of the diffusion node

The memory holds two individuals, A and B, represented by linear machine code. The maximum length of the individuals can be varied between 64-256 words of instructions depending on the application.

The router connects the node to its neighboring nodes, enabling the node to send/read both fitness values and machine code to/from selected neighbors. Please note that every node not only is at the center of its own neighborhood, but also is part of eight other neighborhoods as their north, north-west, west, etc, neighbor.

The CPU is based on the 8-bit PIC16Cxx architecture fitted with an extra integer multiplier and executes all instructions in one cycle [14].

The control block controls the three parts described above and performs the genetic algorithm. The selection is done as a binary tournament between two randomly chosen individuals amongst the ten individuals received by the router (two local and eight neighborhood individuals).

The one-point crossover is done synchronously in a SIMD-like manner by reading two individuals from the router and writing them to the memory. Please note that every node sends a randomly chosen A/B-individual to every node in its neighborhood. It is, however, the receiving node that determines which of the ten received individuals that will participate in the crossover operation, based on the result of the tournament.

The mutation is done at instruction level and randomly replaces one instruction with a randomly generated one.

IV. SIMULATIONS

The design has been simulated both at low level in VHDL to verify its data path and control signals and at a high level with a simulator to verify the algorithm and the representation on real problems.

IV.I. VHDL Simulations

The detailed VHDL simulations have indicated at which frequency the system can operate and the number of gates needed per node.

Given an 8-bit CPU architecture with four general purpose registers, a maximum size of 64 instructions per individual and 16-point regression problems as applications, the architecture is able to evolve 20,000 generations per second. The gate-count per node is around 20,000 gates making it possible to fit a maximum of 500 nodes (1,000 individuals) per FPGA (Virtex XC2V10000). If larger populations are needed, it is very easy to connect several FPGA chips together.

These simulations also showed that the size of a node is not dominated by the CPU but by the memory. The CPU accounts for only about 10 % of the total number of gates per node, yet the evaluation of the individuals is responsible for around 90 % of the execution time. An obvious conclusion would be to integrate two CPU's per node and evaluate both individuals in parallel. This would almost double the performance of the system but only increase the gate count by less than 10 %.

IV.II. Application Simulations

We found the VHDL simulations very encouraging and also wanted to simulate the design, the algorithm and the linear GP representation at a higher level to make sure that the architecture would work on real applications.

A custom made simulator was developed for the diffusion model, allowing the manipulation of its most important parameters, including the topology, neighborhood size, selection algorithm and the linear machine code representation.

During these simulations we used three different regression problems as test problems; the DeJong test-suite function #1 (1), the classic Rosenbrock function (2) and a function suggested by Nordin (3) [13].

$$f(x_1, x_2, x_3) = \sum_{j=1}^{3} x_j^2 \qquad (1)$$

$$f(x_1, x_2) = 100\left(x_1^2 - x_2\right)^2 + \left(1 - x_1\right)^2 \qquad (2)$$

$$f(x_1, x_2) = 5\left(x_1^4 + x_2^3\right) - 3x_1^2 \qquad (3)$$

The functions were resampled in 10 random points every 10 generations and the sum of absolute error in function estimation in these points was used as raw fitness measure. The algorithm described in the implementation section above was used in the experiments as well as the parameters of Table 1.

TABLE I
PARAMETER SETUP

Parameter	Value
Population size (DeJong function)	8,192
Population size (Rosenbrock and Nordin)	18,432
Crossover frequency	70 %
Crossover type	1-point
Mutation frequency	30 %
Selection algorithm	Binary tournament
Maximum code length	64 words

Function set :	ADD W, Fi
	SUB W, Fi
	MUL W, Fi
	MOV W, Fi
	MOV Fi, W
	MOV const, W

Registers	W, F0-F3
Constants	0 .. 31
Maximum number of generations	3,000
Number of runs	100

These high-level simulations were also successful. Averaged over 100 runs, each of the three regression problems (DeJong, Rosenbrock and Nordin) were solved with perfect solutions in 136, 718 and 658 generations respectively. In figure 6, 7 and 8 three example plots from each of the three experiments are shown. The plots show the error of the best individual of the population as a function of the generation number.

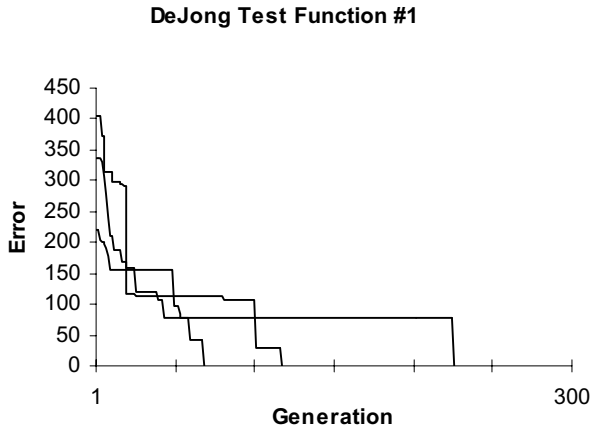

Figure 6 – Three error plots for the DeJong test function problem

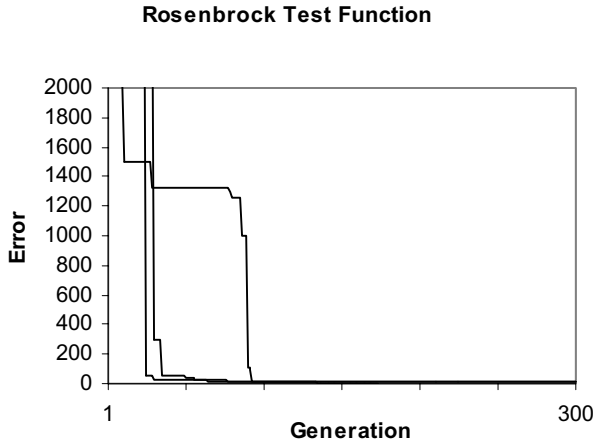

Figure 7 – Three error plots for the Rosenbrock test function problem

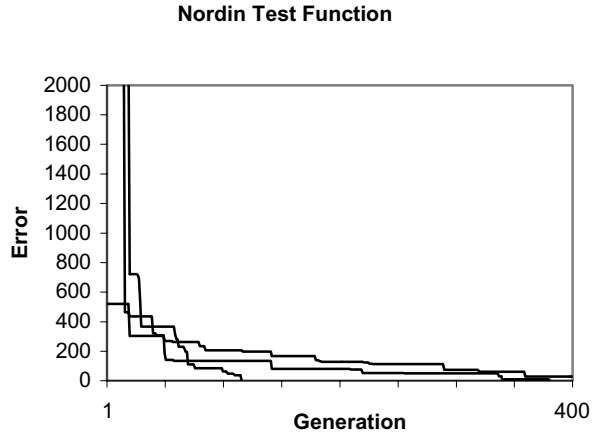

Figure 8 – Three error plots for the Nordin test function problem

V. CONCLUSIONS

We have described a hardware implementation of the massively parallel diffusion model using a linear machine code representation. During both high level simulations and detailed VHDL simulations this design has shown to be efficient on a series of regression problems. Simulations have also indicated that the design can be fine tuned to gain even more performance.

VI. FUTURE WORK

Our custom made simulator for the diffusion model has proven to be a valuable tool in fine tuning the design. We are currently optimizing the design for regression problems using the simulator with different neighborhoods and selection algorithms. Results from these simulations will be available during the spring of 2002. During early summer we plan to do actual FPGA implementations of the refined model based on these simulations.

VII. REFERENCES

[1] Abramson, D, & Abela, J, *"A Parallel Genetic Algorithm for Solving the School Timetabling Problem", In Proceedings of the Fifteenth Australian Computer Science Conference (ACSC-15)*, Volume 14, pp 1-11, 1992.

[2] Abramson, D, Mills, G, & Perkins, S, *"Parallelization of a Genetic Algorithm for the Computation of Efficient Train Schedules", Proceedings of the 1993 Parallel Computing and Transputers Conference*, pp 139-149, 1993.

[3] Banzhaf, W, Nordin, P, Keller, R, Francone, F, *"Genetic Programming - An Introduction", ISBN 1-55860-510-X, pp 330-334, Morgan Kaufmann Publishers Inc*, San Francisco and dpunkt Verlag, Heidelberg, 1998.

[4] Baluja, S, *"A Massively Distributed Parallel Genetic Algorithm (mdpGA)", CMU-CS-92-196R*, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1992.

[5] Bethke, A. D, *"Comparison of Genetic Algorithms and Gradient-Based Optimizers on Parallel Processors : Efficiency of Use of Processing Capacity", Tech. Rep. No. 197*, University of Michigan, Logic of Computers Group, Ann Arbor, MI, 1976.

[6] Cantú-Paz, E, *"Designing Efficient Master-Slave Parallel Genetic Algorithms", IlliGAL Report No. 97004*, University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory, Urbana, IL, 1997.

[7] Cantú-Paz, E, *"A Survey of Parallel Genetic Algorithms"*, Department of Computer Science, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, 1998.

[8] Grefenstette, J. J, *"Parallel Adaptive Algorithms for Function Optimization", Tech. Rep. No. CS-81-19*, Vanderbilt University, Computer Science Department, Nashville, TN, 1981.

[9] Holland, J. H, *"Adaptation in Natural and Artificial Systems"*, The University of Michigan Press, Ann Harbor, 1975.

[10] Kohlmorgen, U, Schmeck, H, Haase, K, *"Experiences with Fine-Grained Parallel Genetic Algorithms"*, Annals of Operations Research, forthcoming.

[11] Koza, J, *"Genetic Programming: On the Programming of Computers by Means of Natural Selection"*, MIT Press, Cambridge, MA, 1992.

[12] Koza, J, Bennett III, F, Shipman, J, Stiffelman, O, *"Building a Parallel Computer System for $18,000 that Performs a Half Peta-Flop per Day", Proceedings of the Genetic and Evolutionary Computation Conference*, pp 1484-1490, 1999.

[13] Nordin, P, Hoffmann, F, Francone, F, Brameier, M, Banzhaf, W, *"AIM-GP and Parallelism", Proceedings of the Congress on Evolutionary Computation*, pp 1059-1066, 1999.

[14] Penfold, R, A, "*Introduction to PIC Microcontrollers*", ISBN 0859343944, 1997.

[15] Schwehm, M, *"Parallel Population Models for Genetic Algorithms"*, Universitat Erlangen-Nürnberg, 1996.

[16] Tanese, R, *"Distributed Genetic Algorithm", Proc. of 3rd Int. Conf. On Genetic algorithms*, pp 434-439, 1989.