Multi-core GE : Automatic Evolution of CPU Based Multi-core Parallel Programs

Gopinath Chennupati
BDS Group
CSIS Department
University of Limerick, Ireland
gopinath.chennupati@ul.ie

R. Muhammad Atif Azad
BDS Group
CSIS Department
University of Limerick, Ireland
atif.azad@ul.ie

Conor Ryan
BDS Group
CSIS Department
University of Limerick, Ireland
conor.ryan@ul.ie

ABSTRACT

We describe the utilization of on-chip multiple CPU architectures to automatically evolve parallel computer programs. These programs have the capability of exploiting the computational efficiency of the modern multi-core machines.

This is significantly different from other parallel EC approaches because not only do we produce individuals that, in their *final form*, can exploit parallel architectures, we can also exploit the same parallel architecture during evolution to reduce evolution time.

We use Grammatical Evolution along with OpenMP specific grammars to produce natively parallel code, and demonstrate that not only do we enjoy the benefit of final individuals that can run in parallel, but that our system scales effectively with the number of cores.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search - Heuristic methods

Keywords

Grammatical Evolution; Multi-cores; Parallel Programming; Symbolic Regression; OpenMP.

1. INTRODUCTION

Real-world applications of evolutionary algorithms (EA) vary dramatically in terms of the computation power required. Depending on the specific problem, techniques varying from caching up to parallel evaluation of both individuals and testcases can be used to reduce computation time.

While there have been some instances where EAs were used to convert existing sequential code to functionally identical parallel code [12], as well as concurrent code [14], there has been no work that produces code that natively exploits multi-core architectures.

It is a well known factor that the execution time limits the emergence of Evolutionary Computation (EC) techniques into the traditional machine learning domain when experimented on large datasets. In fact it is a major concern for any prediction system, for example, if a weather prediction system consumes more than a day or two to produce predictions for the next days then, it is pointless to use that predictor.

The primary reason behind this high computational cost is that the stochastic nature of the EC techniques requires a number of candidate evaluations to produce an optimal solution of certain quality. Genetic Programming (GP) [6] and Grammatical Evolution (GE) [11] are such EAs that require optimization in their computational effort spent on the evaluation of the candidate solutions.

There have been a number of attempts from machine code generation to the usage of parallel computational resources that address this issue. Graphics Processing Units (GPUs) attracted a lot of attention from the EC community with their high computational capabilities through light weighted cores. One such work related to GP is [10] that tried to reduce the evaluation time while it also questions the high performance speedups reported in the other GPU based EC implementations. Another attempt in [4] tried to reduce the execution time through the termination of the runs that produce low quality solutions.

However, recent desktops and PCs are made of multiple cores that have better compute capabilities to improve the speedup at a reasonable cost. These multi-core processors are equipped with a shared memory configuration that can be accessed by all the cores which in turn helps to achieve efficient parallelization. Little work has been done on exploiting the compute capabilities of these architectures. However, utilizing the power of the multi-cores can optimize the execution time significantly.

This paper proposes to automatically evolve parallel computer programs that have the ability to exploit the compute capabilities of the modern CPU based multi-core architectures. We use OpenMP thread based programming practices along with GE grammars for the automatic evolution of these parallel programs. This approach is assessed on two standard bench mark problems: 6-multiplexer and a sextic polynomial regression.

2. BACKGROUND

GE is an evolutionary algorithm that can evolve computer programs in an arbitrary language through a linear genome representation and the rules of the BNF grammars. Each individual in the GE population is a variable length binary string and, typically, the genome is divided into 8 bit *codons*,

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.

http://dx.doi.org/10.1145/2598394.2605670.

which are used to select the possible rules of the grammar. GE *mapping* is a crucial component of its evolutionary cycle which is responsible for producing code in any programming language, and is an extra step which does not occur in GP.

2.1 Phenotype Evaluation

A general consensus in the EC community is that an EA's fitness evaluation is the most computationally expensive task. Particularly in GE, experimental observations presented in [3] confirm this statement (with 66% of total execution time) while GE mapping (with 33% of total execution time) is the next computationally expensive task. GE is unique in its fitness evaluation as it is calculated through the evaluation of phenotype whereas all the remaining genetic operations (such as crossover, mutation) are performed on the genotype. This separation between the solution and the search space has been made possible with the help of the Context Free Grammars (CFGs) and the GE specific *mapping* process as explained here.

GE *mapping* process converts a genotype into phenotype, a computer program in a user specified language. GE first decodes a variable length binary string to a string of eight bit integers term known as *codons*. These codons are then used to pick rules from a CFG with this function:

Rule = (Codon Integer Value) % (# rules for this non-terminal)

Most often (not always), the fitness of the phenotype is evaluated using a complier, an *evaluator* in GE. It evaluates the evolved computer program serially over all the fitness cases. One drawback with the current GE system is that it can only use the compute capabilities of only one core that leaves the remaining cores of a multi-core machine idle. In this paper we focus only on optimizing the execution time of GE phenotype evaluations that can make use of the computational power of these modern machines.

2.2 Related Work

Most of the EC approaches (GA, GP) are inherently parallel or contain the design that is suitable for explicit parallelization. The explicit parallelization approaches can largely be classified as *population parallel* and *data parallel* as shown in [8]. The former deals with a population of individuals with different sets of configurations that exploit the power of multiple computers, while the latter deals with simultaneous data computations. Cantú-Paz [2] contemplated different parallelization configurations in GAs whereas [8] (Chapter 10) listed out different approaches that can accelerate GP. However, recently graphics processing units (GPUs) emerged as highly parallel computing resources. Robilliard et al. [10] tried to exploit the power of G80 GPUs in paralleling GP fitness evaluations. Chitty [5] demonstrated a population parallel approach that fully exploits the power of CPU based multi-cores on a new implementation of GP.

Along these EC approaches, also the design of GE is "embarrassingly parallel" that takes advantage from the separation offered through its mapping. The first parallelization attempt on GE was made in [13] with an island model that was implemented in python programming language. It optimized the total execution time when tested on a 17 node cluster with each node containing the Intel Pentium IV Dual Core processor. Osmera et al. [7] proposed a two level population parallel approach with GE in the first level and Differential Evolution in the next level in a hierarchical master-slave configuration, where the slaves port their best candidate solutions to the master. They tested it on a group of 6 computers networked together with one master and five slaves with two populations (female, male) on each one.

Recently, for the first time [9] reported promising acceleration in the execution time by the entire GE algorithm on GPU cores. Although the proposed approach optimized the execution time, they also have reported a serious limitation on the use of GPU memory that would decelerate the speed of execution as the population size scales up.

Parallel GE implementations are relatively few in number due to the need to access specialized parallel hardware such as GPUs. However, modern CPU vendors (Intel i7) are offering desktop computers with multiple cores with each core having the power of a standard single core processor. The computational power of these architectures has been little explored leaving a lot of scope for us to investigate. Despite the parallelization investigations in the literature, to the best of our knowledge for the first time we propose to exploit the compute capabilities of multi-core architectures through an automatic evolution of parallel programs through grammars. In fact the automatic evolution of parallel programs frees us from the shackles of maintenance costs of parallelization that parallel programming researchers often face.

```
<start> ::= <omp_pragmas><for_loop><data>
  <evolved_expr>
<omp_pragmas> ::=  #pragma omp parallel
  shared(Evolved, chunk) private(i) {
  #pragma omp for schedule (dynamic,chunk)
<for_loop> ::= for(i=0; i<FITNESS_CASES;
  i=i+1) {
<data> ::= xvals[i]; yvals[i];
<evolved_expr> ::= temp= <expr>;<assign>
<expr> ::= mymul(<expr>, <expr>) |
  mysub(<expr>, <expr>) |
  myadd(<expr>, <expr>) |
  pdiv(<expr>, <expr>) |
  if_cond(<expr>, <expr>, <expr>, <expr>)
  | (<expr>) | sin(<expr>) | cos(<expr>)
  | tan(<expr>) | exp(<expr>) | xvals[i]
  | yvals[i] | 1.000 | <const> | -<const>
<const> ::= 0.<digit><digit><digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 |
  7 | 8 | 9
<assign> ::= if(temp < 0) { Evolved[i] = 0;
  } else { Evolved[i] = 1; } }
```

Figure 1: Design of multi-core grammars

3. MULTI-CORE GE

Each core of a multi-core machine is a single unique CPU. Utilizing the compute capabilities of these cores is relatively a simple process in comparison with the GPU cores. Creating different threads on each core enables us to exploit the power of these cores. Thus, under any operating system environment, calling OpenMP enabled C/C++ function creates user requested number of threads. OpenMP offers parallel programming *pragmas* that can execute independent threads on cores.

We use these OpenMP *pragmas* to automatically evolve parallel programs that are compatible with multi-core CPU shared memory architectures. Figure 1 presents the grammars that are used in the automatic evolution of parallel programs. For these preliminary investigations, the evolved programs follow a *data parallel* approach so that at any given period of time only one phenotype is evaluated across multiple fitness cases in parallel.

In this approach, load balancing is a serious concern that was caused by thread scheduling. OpenMP improves this inefficiency with the help of a few thread scheduling options, *dynamic scheduling* is the best option among them. It dynamically assigns t iterations of the remaining iterations to the threads that are idle, finding the optimal t can be hard some times. That means the available data is divided into chunks of size t and allocates each chunk dynamically to the threads immediately upon completion of their execution.

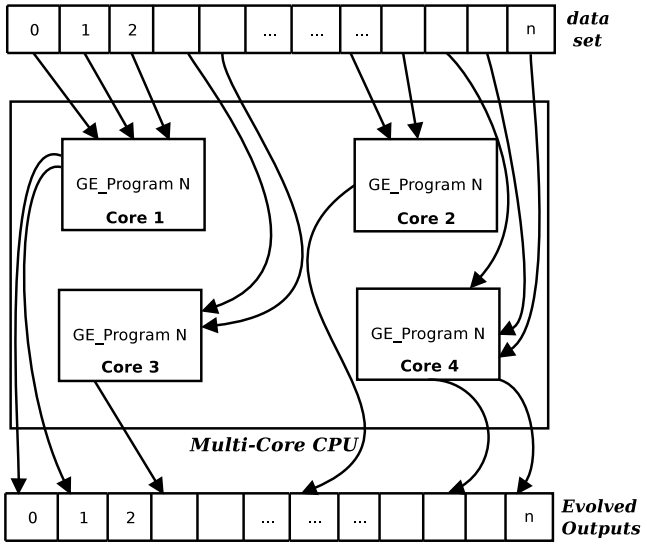

Figure 2: Parallel evaluation scheme of the evolved GE program on a multi-core CPU.

Figure 2 shows the phenotype evaluation scheme across multiple cores of a modern CPU. The main characteristic of this approach is, a **single** GE program is executed across all the cores with a different set of fitness cases, thus making the execution single instruction multiple data (SIMD). That is fitness cases are processed in parallel across all the cores. For example on a 4 core machine, every core evaluates $1/4^{th}$ of the fitness cases (this may vary due to dynamic scheduling) ideally and must report an evaluation speedup factor of 4.

In this scheme some threads finish their execution early while the others fall behind so that we end up with an issue of thread synchronization in calculating the final fitness value. In such instances the OpenMP *parallel reduction* is applied on squared error (regression problems), the number of hits (classification problems) in a separate function. Since both the operations involve summation that can be generalized as the sum of n numbers for which the time complexity is $O(n^2)$. With the reduction operation the complexity has been reduced to $O(nlogn)$. In reduction there is a chance for the occurrence of *race* condition when the evolved class values are compared with that of the target (*if (evolved[i] == target[i])*) where the evolved class in thread 1 might race to check the equivalence with thread 2 of the target class. In order to prevent this an OpenMP *critical section* was used where only one thread can execute at any given time in that region.

4. EXPERIMENTS

In order to test the proposed approach the experiments were carried out on a cluster node with an Intel (R) Xeon (R) CPU E7-4820 with a processor speed of 2 GHz. It contains 16 cores each of which are enabled by hyper threading with an ability to run 2 independent threads per core and 18 Mb of L3 cache memory was shared among all the cores. The experimental parameters were set as follows. The population was initialized randomly with the maximum depth set to 25, while the minimum depth was 15. It is interesting to see the performance of the proposed approach with various population sizes and a variable number of CPU cores. Hence, the population sizes were kept at $25, 50, 100, 200, 400$ while the number of CPU cores were at $2, 4, 8, 16$. One point crossover with a probability 0.9, point mutation with a probability 0.01 were used. We used steady state GA where the best individuals replace the worst in the population. GNU GCC evaluator is used for phenotype evaluations. A total of 30 runs were conducted with a generation count of 50.

4.1 Results

We evaluate the performance of our approach on two standard benchmark problems: 6-multiplexer and, sextic polynomial regression ($x^6 - 2x^4 + x^2$). The first benchmark contains 64 training points with 2 address bits (A0-A1), 4 data bits (D0-D4) and the second benchmark is a regression problem that also has 64 training points in the range [-1.5,+1.5]. A complete description of the serial implementation of the grammars can be found in [1]. We focus on our goal of optimizing the phenotype evaluation time and report those results, despite the fact that there is no significant difference in the serial and the parallel (proposed) version of the solutions. The total execution time results were measured using the OpenMP platform independent timer utility function.

Table 1 demonstrates the performance of multi-core GE. The average evaluation time of fitness evaluation results of the evolved parallel programs with varying number of CPU cores is shown over 30 runs on both the bench mark problems. On 6-multiplexer problem, with 2 cores there is no significant difference in the average evaluation time whereas for the remaining cores $(4, 8, 16)$ it was reduced significantly recording better speedups. For sextic symbolic regression, incrementing the cores to 2 has in fact increased the average evaluation time as it a well known OpenMP pattern. Except that, for the remaining configurations significant reductions were observed. Overall, the performance of the multi-core parallel programs reported promising results in the fitness evaluations.

5. CONCLUSION AND FUTURE WORK

In this paper we presented a new method to exploit the power of modern multi-core machines through an automatic evolution of parallel programs that can reduce the risk of maintaining the parallel programming. Modern multi-cores are more suitable for multiple instruction multiple data (MIMD) style of execution, whereas the proposed approach follows SIMD, following multi-core adaptable style of execution might

Table 1: Average evaluation time (in secs) results (average [standard deviation]) of multi-core GE over 30 runs on 6-multiplexer and sextic regression problems.

Problem	Pop Size	Cores				
		1	2	4	8	16
6-multiplexer	25	88.49 [11.86]	92.78 [6.01]	41.54 [6.45]	22.01 [6.82]	12.6 [6.41]
	50	226.29 [24.65]	173.26 [12.31]	123.65 [16.52]	46.56 [10.47]	30.13 [15.23]
	100	347.13 [25.40]	335.84 [27.32]	177.11 [24.81]	69.56 [21.92]	44.16 [21.95]
	200	703.21 [35.42]	698.76 [19.14]	285.86 [12.61]	185.54 [17.32]	86.28 [17.92]
	400	1399.54 [15.95]	1428.75 [28.42]	897.14 [23.36]	392.03 [26.70]	169.85 [33.51]
sextic regression	25	107.96 [17.71]	152.07 [25.44]	36.86 [20.60]	17.27 [21.06]	10.77 [20.88]
	50	217.69 [26.52]	279.65 [21.68]	82.86 [26.63]	35.03 [33.14]	22.62 [32.94]
	100	482.91 [34.37]	601.77 [33.43]	183.88 [24.82]	80.28 [29.78]	55.54 [29.32]
	200	847.17 [26.14]	1361.33 [37.93]	498.85 [22.50]	168.64 [20.41]	118.35 [17.01]
	400	1748.67 [33.87]	3757.92 [27.34]	1030.70 [16.24]	405.21 [18.41]	261.86 [18.93]

report much better performance with optimal efficiency. Recall that we evaluated the final fitness of the phenotype in synchronous fashion using a *critical* section where some threads wait for the others to finish their execution. Although we observed improvement in the speedups this synchronous execution makes the cores to sit idle as a result CPU time is underutilized. Asynchronous phenotype evaluation is one of the recommended directions to investigate.

It is interesting to see the length of the genotypes as it is clear from the presented grammars that the size of the grammars is larger than the serial version for the respective problems. There is a chance to get longer individuals that are being evaluated in less time. It is also possible to investigate by incrementing the size of the fitness cases, using a generational strategy as opposed to the used steady state replacement procedure that which might pose different challenges with an increased breeding time.

Experimenting with highly decomposable problems such as quick sort, vertex colouring and the ones that also involve task level parallelism is another interesting research direction.

6. REFERENCES

[1] R. M. A. Azad and C. Ryan. An examination of simultaneous evolution of grammars and solutions. In T. Yu, R. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice III*, volume 9, pages 141–158. Springer, US, 2006.

[2] E. Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10(2):141–171, 1998.

[3] G. Chennupati, C. Ryan, and R. M. A. Azad. An empirical analysis through the time complexity of GE problems. In R. Matousek, editor, *19th International Conference on Soft Computing, MENDEL'13*, pages 37–44, Brno, Czech Republic, Jun, 26-28 2013.

[4] G. Chennupati, C. Ryan, and R. M. A. Azad. Predict the performance of an evolutionary algorithm run. In *the Genetic and Evolutionary Computation Conference*, GECCO '14, Vancouver, BC, Canada, Jul, 12–16 2014, to appear.

[5] D. M. Chitty. Fast parallel genetic programming: multi-core cpu versus many-core gpu. *Soft Computing*, 16(10):1795–1814, 2012.

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

[7] P. Osmera, O. Popelka, and P. Pivonka. Two level parallel grammatical evolution. In *IEEE World Congress on Computational Intelligence, IEEE Congress on Evolutionary Computation, CEC 2008*, pages 2473–2480, June 2008.

[8] R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.

[9] P. Pospichal, E. Murphy, M. O'Neill, J. Schwarz, and J. Jaros. Acceleration of grammatical evolution using graphics processing units. In *Proceeding of Genetic and Evolutionary Computation Conference*, GECCO '11, pages 431–438. ACM, 2011.

[10] D. Robilliard, V. Marion, and C. Fonlupt. High performance genetic programming on GPU. In *Proceedings of the Bio-inspired algorithms for distributed systems*, pages 85–94. ACM, 2009.

[11] C. Ryan, J. J. Collins, and M. O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 83–95. Springer, 1998.

[12] C. Ryan and P. Walsh. The evolution of provable parallel programs. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 295–302, Stanford University, CA, USA, 1997. Morgan Kaufmann.

[13] M. Stallard. A parallel approach to grammatical evolution in python, 2006.

[14] A. Trenaman. Concurrent genetic programming, tartarus and dancing agents. volume 1598 of *LNCS*, pages 270–282. Springer, 1999.