Automatic Lock-free Parallel Programming on Multi-core Processors

Gopinath Chennupati, R. Muhammad Atif Azad and Conor Ryan
Bio-Computing and Developmental Systems Group
Computer Science and Information Systems Department
University of Limerick, Ireland
Gopinath.Chennupati@ul.ie, Atif.Azad@ul.ie, Conor.Ryan@ul.ie

***Abstract*—Writing correct and efficient parallel programs is an unavoidable challenge; the challenge becomes arduous with lock-free programming. This paper presents an automated approach, *Automatic Lock-free Programming (ALP)* that avoids the programming difficulties via locks for an average programmer. ALP synthesizes parallel lock-free recursive programs that are directly compilable on multi-core processors. ALP attains the dual objective of evolving parallel lock-free programs and optimizing their performance. These programs perform (in terms of execution time) significantly better than that of the parallel programs with locks, while they are competitive with that of the human developed programs.**

***Keywords*-Multi-cores; Lock-Free Programming; Evolutionary Computation; Program Synthesis; OpenMP**

I. Introduction

As multi-core processors become the norm, it is compelling to develop parallel programs that *efficiently* coordinate processing shared objects. Although powerful, parallel programming is a non-trivial task, as sequential developers still struggle with the parallel design issues such as synchronization, locks, and correctness. For an average programmer, it is difficult to trade-off execution time and these challenges.

Compilers assist programmers in optimizing time through a set of transformations while preserving the correctness of the original code. However, performance of the evolved parallel programs has superseded that of the compiler optimizations [4], where compilation times were long.

Typically, *locks* ensure mutual exclusion among the shared objects; however, while locks assure program correctness, performance of an application degrades when the contention for the shared resource is high, or when it is used often.

Therefore, avoiding locks is important [11], while ensuring correct lock-free programs is often challenging even for the experts. Lock-free programs rely on two operations: first, atomic instructions such as *read-modify-write*, and second, memory barriers such as *load* and *store*. This paper focuses on atomic operations, where thread waiting due to locks is eschewed to improve the performance of parallel programs.

We introduce Automatic Lock-free Programming (ALP), where the goal is to *automatically* generate **natively** parallel *recursive* programs that avoid *locks* on shared resources. In fact, recursion is an interesting problem with its complexity in achieving atomicity, particularly when the recursive calls are executed simultaneously. Thus, we automatically produce the first natively parallel lock-free recursive programs with as little human intervention as possible. It helps to minimize a few parallel design difficulties of sequential programmers.

Grammatical Evolution (GE) [20] is used to automatically generates program semantics. GE generates a *population* of programs that obey the syntactic constraints of a pre-specified Context Free Grammar (CFG). We evaluate ALP on five standard benchmark recursive programs. The results report an average speed-up of 10.43 with a significant improvement of 25.21% over the evolved parallel programs. The performance of the automatically generated lock-free programs is competitive with that of the human written programs.

The remainder of the paper is organized as follows: section II briefly describes the preliminaries required for the proposed approach; section III outlines ALP; The experimental approach and results are in section IV, while section V analyses the proposed approach; and finally section VI concludes and recommends future directions.

II. Preliminaries

We briefly describe lock-free programming (section II-A) and a motivating example (section II-B) that describes the problem of locks. Then, we delve into the literature (section II-C) of lock-free programming and genetic improvement (section II-D).

A. Lock-Free Programming

Lock-free programming uses multiple threads to operate on shared data, while refraining the threads from blocking each other. The two major tools in lock-free programming are: atomic and reordering operations.

First, atomic operations ensure that the execution of one thread is invisible to other threads. Because if those threads access the half modified values of the thread under execution, then this results in an inconsistent state. It is important to avoid such errors in lock-free programming. Second, reordering the memory reads and writes, when they do not occur in the order that they are written in a program. For example, sometimes, a thread writes the shared data that has not been updated by other threads, similarly, a thread reads the shared data which is not up to date. The challenge then is to use these operations in writing the lock-free programs. However, in this paper, we focus only on the atomic *read-modify-write*

978-1-5090-0623-6/16/$31.00

operations on multi-cores. The following section sets the context with an example.

B. Motivating Example

Fig. 1 implements a parallel recursive Fibonacci algorithm in C. It illustrates the importance of acquiring and/or releasing locks on shared objects.

```
1/* Fibonacci with parallel recursion */
 int fib(int n) {
3  int i, j;
   if (n <= 2)  return n;
5  else  {
    #pragma omp parallel sections \
7   shared(i, j, res) private(n)
    { //more than one thread in operation
9     #pragma omp section
      { // update result with first call
11      #pragma omp critical
        { i = fib(n-1); res += i; }
13    }
      #pragma omp section
15    {// update result with second call
        #pragma omp critical
17      { j = fib(n-2); res += j; }
      } return res;
19} } }
```

Fig. 1: Motivating example: *Fibonacci* program in OpenMP.

The two recursive calls are executed in parallel, where the result of each call is saved in the temporary variables (i, j). To assure mutual exclusion among threads, *OpenMP critical section* stores the intermediate recursive calls and updates the end result (*res*) while it also synchronizes the communication among threads. The critical construct allows only one thread to execute the region of the code (lines 11, and 16) that it encircles. Under-the-hood, the critical directive locks the shared data to the thread in the region and restricts the other threads from accessing it. As a result, the program fails to realize the true potential of the multi-core processor. It is necessary to avoid such locks for better performance. Next, we discuss the literature of lock avoidance.

C. Lock-free Programming Literature

A notable attempt to avoid locks by Lamport [15] allowed single writer and multiple readers on shared variables. This accelerated interest in implementing lock-free data structures. Herlihy [13] used the *compare-and-swap* atomic primitive in the non-blocking implementation of linked lists. These lists suffered from poor performance due to the centralized swapping behaviour of the primitive. Thereafter, Valois [26] presented an effective *compare-and-swap* based non-blocking implementation of linked lists, however, it was an erroneous implementation that were addressed later by Harris [11].

Desnoyers and Dagenais [9] presented a lockless buffering scheme for kernel tracing on multi-cores. They improved the performance through the use of local atomic operations that operate on local CPU variables. Dechev et al. [8] introduced a novel *lockless containers/data concurrency* library to design non-blocking algorithms on multi-cores. Tordini et al., [25] proposed non-blocking high-level programming patterns that help programmers omit the low-level details about synchronization and memory management. Later, Al-Bahra [1] explored different alternatives for lock-based synchronization to improve the scalability of non-blocking algorithms on multi-cores. It showed that the *read-modify-write* operations are expensive due to their reliance on underlying programming environment and architecture.

However, lock-free programming requires the knowledge of both hardware and compiler. Since, even the programming gurus proposed erroneous [11], [26] lock-free algorithms, adept programming skills are necessary for writing lock-free programs. Recently, Timnat and Petrank [24] automatically transformed the lock-free data structures to efficient wait-free data structures. In the same spirit, we leverage GE for lock-free programming; work in literature, Multi-Core Grammatical Evolution (MCGE-II) [5] explored the viability of GE for parallel programming, albeit using locks. Evolutionary attempts like this, fit rightly into the recently emerging branch of *genetic improvement* of software.

D. Genetic Improvement of Software

Although genetic programming was in use in improving the existing software [22], it has come to the fore with the works of fixing the bugs [28] in legacy code. Similarly, recent attempts [10], [18], [21] improve the functional and non-functional properties of an existing software, commonly known as *genetic improvement*.

A few interesting genetic improvement attempts include, Álvarez et al., [2] applied GE in optimizing the cache memory access patterns in embedded systems. Cody-Kenny et al., [7] employed GP to improve the performance of Java programs by reducing the number of Java byte code instructions being executed. A discussion on the genetic improvement of existing software can be seen in [16].

Some of these attempts try to preserve the semantic equivalence of the original programs through the co-evolved test cases [3], [29] while others maintain it [17], [19] by generating the test cases from running the original programs.

In contrast to the genetic improvement methods in literature, ALP synthesizes lock-free parallel recursive programs. The resultant source code is directly compilable on multi-core processors. Since ALP does not rely on an input program, there is less difficulty in preserving the semantic equivalence. Nonetheless, the hardship transforms into evolving correct parallel lock-free programs that perform efficiently on multi-cores. For program correctness, we generate test cases (similar to [19]) from the original programs.

III. Automatic Lock-free Programming

Automatic Lock-free Programming (ALP), evolves parallel lock-free recursive programs. The resultant source code is directly compilable on multi-core processors. In realising this

⟨*program*⟩ ::= ⟨*condition*⟩ ⟨*parcode*⟩

⟨*condition*⟩ ::= if(⟨*input*⟩ ⟨*lop*⟩ ⟨*const*⟩) '{' ⟨*newline*⟩ ⟨*tab*⟩ ⟨*line1*⟩; ⟨*newline*⟩ ⟨*tab*⟩ ⟨*lockfree*⟩ ⟨*line2*⟩ ⟨*newline*⟩ '}'

⟨*parcode*⟩ ::= else '{' ⟨*newline*⟩ ⟨*omppragma*⟩ ⟨*private*⟩ ⟨*shared*⟩ ⟨*blocks*⟩ ⟨*newline*⟩ '}' ⟨*newline*⟩ '}' ⟨*newline*⟩ ⟨*result*⟩

⟨*omppragma*⟩ ::= ⟨*ompdata*⟩ | ⟨*omptask*⟩

⟨*ompdata*⟩ ::= #pragma omp parallel | #pragma omp parallel for

⟨*omptask*⟩ ::= #pragma omp parallel sections | #pragma omp task

⟨*shared*⟩ ::= shared(res) ⟨*newline*⟩ '{'

⟨*private*⟩ ::= private(⟨*input*⟩) | firstprivate(⟨*input*⟩) | lastprivate(⟨*input*⟩)

⟨*blocks*⟩ ::= ⟨*parblocks*⟩ | ⟨*blocks*⟩ ⟨*newline*⟩ ⟨*blocks*⟩

⟨*parblocks*⟩ ::= ⟨*secblocks*⟩ | ⟨*taskblocks*⟩

⟨*secblocks*⟩ ::= #pragma omp section ⟨*newline*⟩ '{' ⟨*newline*⟩ ⟨*tab*⟩ ⟨*line1*⟩; ⟨*newline*⟩ ⟨*tab*⟩ ⟨*lockfree*⟩⟨*line2*⟩ ⟨*newline*⟩ '}'

⟨*taskblocks*⟩ ::= #pragma omp task ⟨*newline*⟩ '{' ⟨*newline*⟩ ⟨*tab*⟩ ⟨*line1*⟩; ⟨*newline*⟩ ⟨*tab*⟩ ⟨*lockfree*⟩⟨*line2*⟩ ⟨*newline*⟩'}'

⟨*lockfree*⟩ ::= __sync_fetch_and_add | __sync_fetch_and_sub | __sync_add_and_fetch | __sync_sub_and_fetch

⟨*line1*⟩ ::= int a = ⟨*expr*⟩;

⟨*line2*⟩ ::= (&*res*, a);

⟨*expr*⟩ ::= ⟨*input*⟩ | ⟨*stmt*⟩ | ⟨*stmt*⟩ ⟨*bop*⟩ ⟨*stmt*⟩

⟨*stmt*⟩ ::= fib(⟨*input*⟩ ⟨*bop*⟩ ⟨*const*⟩);

⟨*lop*⟩ ::= '>=' | '<=' | '>' | '<' | '=='

⟨*bop*⟩ ::= + | - | * | /

⟨*const*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨*result*⟩ ::= return res;

⟨*input*⟩ ::= n

⟨*newline*⟩ ::= *n*

⟨*tab*⟩ ::= *t*

Fig. 2: Multi-core grammar to automatically generate a parallel lock-free recursive Fibonacci program.

objective, we extend the existing framework MCGE-II [5] in literature. Note that MCGE-II uses GE as an underlying search engine and OpenMP pragmas to produce parallel programs. MCGE-II generated programs exhibit the locking behaviour shown in Fig 1. ALP avoids these locks automatically while ensuring program correctness and optimizes the execution time. Next, we describe the design of ALP grammar (section III-A) and fitness evaluation (section III-B).

A. Grammar Design

ALP avoids locks in the evolving parallel programs through the use of GNU GCC built-in atomic primitives [23]. The design of the GE grammars contain problem specific information, OpenMP pragmas, and the atomic primitives.

Fig. 2 shows the ALP grammar to generate lock-free Fibonacci programs. It defines a compilable solution space of parallel lock-free programs, given which, GE searches for an optimal solution that passes all the tests in a given dataset. The program accepts an integer (*n*) input (`<input>`), while *res* returns the result. Note *res* is a global variable initialized to zero.

The non-terminal `<omppragma>` allows to choose between OpenMP task or data parallel directives, and further specialises with a choice between `<taskblocks>` and `<secblocks>`. The non-terminal `<taskblocks>` refers to the *task* work-sharing construct while `<secblocks>` refers to the *section* construct. The non-terminals `<private>` and `<shared>` map to thread-private (*n*) and thread-shared (*res*) variables respectively. Here, GE prefers to select one of the three OpenMP private clauses.

The non-terminal `<lockfree>` refers to the built-in GCC atomic memory access primitives. Thus, the evolution suitably selects from these atomic *read-modify-write* operations for addition (`__sync_fetch_and_add`, `__sync_add_and_fetch`) and subtraction (`__sync_fetch_and_sub`, `__sync_sub_and_fetch`), which returns either an old or a new value as defined. The `<expr>` returns an integer value or a recursive call (`<stmt>`) of the program.

The use of an inapt directive and/or an atomic primitive degrades the fitness of the evolved program. Those of which fail to compile are assigned a zero fitness, while others report a relatively low fitness. Next, we discuss the fitness evaluation of ALP evolving programs.

B. Fitness Evaluation

The execution time of the automatically generated programs depend primarily on the choice of parallelization pragmas, and the atomic primitives. The goal is to generate a correct lock-free parallel recursive program that minimizes the execution time. Although the use of any OpenMP pragma will result in a parallel program, the execution time changes significantly among those pragmas. To assure the right degree of parallelism, we consider execution time of the evolved program in its fitness evaluation.

TABLE I: Summary of the benchmark programs under investigation with their properties. The 'Local Variables' column represents the number of temporary variables used in the respective program.

#	Program	Type		Program Description	Local Variables (LV)	Range
		Input	Return			
1	Sum-of-N	int	int	sum of first N numbers	3	[1, 1000]
2	Factorial	int	unsigned long long	product of all the positive integers less than or equal to input N	3	[1, 60]
3	Fibonacci	int	unsigned long long	example from section II-B	3	[1, 60]
4	Binary-Sum	int [ ], int, int	int	sum of adjacent elements of an array	2	[1, 1000]
5	Reverse	int [ ], int, int	void	reverse the elements of an array	2	[1, 1000]

Thus, the fitness function (f_{pprog}) is:

$$f_{pprog} = \frac{1}{\left(1+t\right)} * \frac{1}{\left(1+\frac{1}{N}\sum_{i=1}^{N}|y_i-\hat{y}_i|\right)} \quad (1)$$

where, t is the execution time of the program over N data points; y_i and $\hat{y}_i$ are the desired and the evolved outputs of a program respectively. In eq. 1, the first term *normalized execution time* spurs to select an optimal OpenMP pragma and/or an atomic primitive, while the second term *normalized mean absolute error* helps to solve the problem. Thus, the fitness function promotes both efficient parallelization and correctness of a program.

IV. Experiments

We report experiments that are designed to: **1) evaluate the performance** of automatically generated lock-free benchmark recursive programs over their serial counterparts; and **2) measure scale-up** of the automatic lock-free programs over the state-of-the-art methods.

A. Experimental Setup

We evaluate ALP on five benchmark recursive programs in C, all of which vary in their complexity. Table I presents the summary of these programs. The programs accept an input from 30 instances dataset. The programs *Sum-of-N*, *Factorial*, *Fibonacci* accept single integer, while *Binary-Sum*, and *Reverse* accept an array of 1000 integers as input. The 'Range' column specifies a closed interval from which the inputs are generated randomly. The 'Type' column shows the *input* and *return* types, where *Input* inferences the number of arguments. The *Return* changes according to the program due to the limitations in the data type range in C. The 'Local Variables' column refers to the number of temporary variables used in the respective program.

1) Parameters: Our algorithm has several parameters that potentially have a significant contribution towards automatic generation of quality source code. GE uses the default parameters: Table II lists those parameters as well as the hardware and software specifications used in the experiments. Each individual encodes a derivation tree with the depth of the trees ranging in between 9 and 25. We execute GE for 50 runs over 100 generations.

TABLE II: GE parameters and experimental environment.

Parameter	Value
Initialization	Random
Strategy	Steady State
Initial Minimum Depth	9
Initial Maximum Depth	25
Number of Generations	100
Population Size	500
Crossover Probability	0.9
Mutation Probability	0.1
Number of Runs	50
Hardware & Software Specifications	
CPU	Intel (R) Xeon (R) E7-4820
Cores	{1,2,4,8,16}
Operating System (OS)	Debian Linux v 2.6.32, 64-bit
C++	GNU GCC v 4.4.5
GE Framework	libGE [**?**] v 0.26
OpenMP	libgomp v 3.0
Timer utility	*omp_get_wtime*()

2) Experimental Environment: The experiments are conducted on a 2 GHz Intel Xeon processor running a 64-bit Debian OS. The number of cores are doubled starting from 1 till 16. We use libGE, an open source C++ implementation of GE. In general, the fitness is evaluated by either compiling or interpreting the GE generated programs. In this paper, the evolved programs are compiled with GCC compiler with *-fopenmp* flag. The execution time of the programs is calculated with OpenMP timer utility, *omp_get_wtime*().

B. Experimental Results

The results are two-fold: first, we compare the *average execution time (AET)* of the lock-free programs with that of the serial counterparts; second, we compare the speed-up of lock-free programs with that of the MCGE-II evolved parallel programs that contain locks.

1) Average Execution Time: AET is the average execution times of the best-of-run programs across all the runs, and is measured as shown in Eq. 2:

$$AET = \frac{\sum_{r=1}^{R} T_{bprog}(r)}{R} \quad (2)$$

where $T_{bprog}(r)$ is the execution time of the best program in a given run r, while R is the number of runs.

Table III compares the average execution time of automatically generated lock-free programs (on different number of

TABLE III: Average execution time (in secs) (AET [standard deviation]) of all the programs with different number of cores.

Program	Cores					Standard GE (serial programs)
	1	2	4	8	16	
Sum-of-N	4285.31 [75.95]	4607.86 [58.42]	1518.75 [43.36]	853.92 [46.70]	483.23 [43.51]	4099.54 [53.18]
Factorial	3483.48 [84.71]	3745.67 [67.26]	1175.74 [71.62]	631.61 [58.43]	415.06 [68.14]	3247.54 [74.44]
Fibonacci	3796.30 [52.16]	2456.19 [55.46]	1194.52 [44.04]	623.46 [38.07]	404.97 [38.39]	4150.93 [86.72]
Binary-Sum	2223.29 [33.87]	1631.53 [47.34]	1073.12 [20.34]	321.53 [28.41]	193.08 [33.51]	2571.14 [26.98]
Reverse	3336.69 [27.36]	3467.05 [24.05]	1539.66 [23.97]	521.23 [27.56]	275.36 [38.92]	3328.35 [53.57]

Cores	Program	Wilcoxon Signed Rank Sum Test			A measure
		Rank Sum	p value	Significant	
2	Sum-of-N	2297	0.7819	No	–
	Factorial	2198	0.2883	No	–
	Fibonacci	2821	0.5119	No	–
	Binary-Sum	3179	0.9178	No	–
	Reverse	2079	0.1178	No	–
4	Sum-of-N	2043	0.0426	Yes	0.4545
	Factorial	2793	0.0028	Yes	0.6151
	Fibonacci	3253	0.0061	Yes	0.8259
	Binary-Sum	2529	0.0351	Yes	0.6815
	Reverse	2471	0.0478	Yes	0.4851

Cores	Program	Wilcoxon Signed Rank Sum Test			A measure
		Rank Sum	p value	Significant	
8	Sum-of-N	2189	0.0338	Yes	0.5083
	Factorial	2815	0.0068	Yes	0.5917
	Fibonacci	3321	0.0032	Yes	0.7392
	Binary-Sum	2079	0.0017	Yes	0.8559
	Reverse	2479	0.0318	Yes	0.5158
16	Sum-of-N	1952	0.0426	Yes	0.4834
	Factorial	2409	0.0021	Yes	0.6255
	Fibonacci	3253	0.0046	Yes	0.8919
	Binary-Sum	2221	0.0051	Yes	0.6156
	Reverse	2008	0.0047	Yes	0.5794

Fig. 3: Significance tests (at $\alpha = 0.05$) show that the ALP outperforms serial programs on all the cores except 2. "Yes" states the results are significant and "No" states that they are insignificant. A measure shows the probability at which, ALP is better over the other.

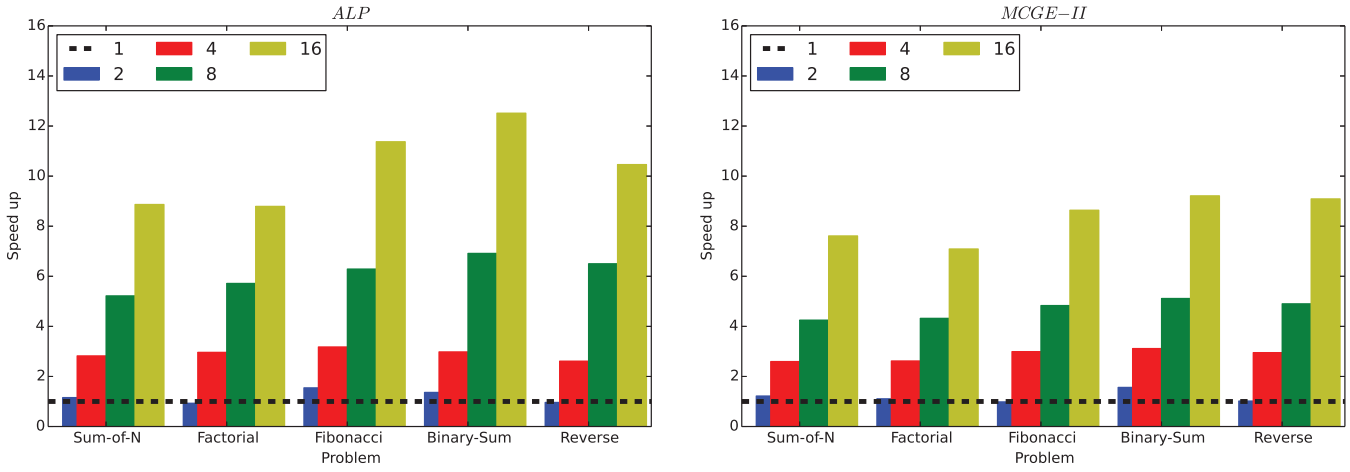

Fig. 4: Speed-up in average execution time (AET) of ALP (left) versus MCGE-II (right) at different number of cores.

Cores	Program	Wilcoxon Signed Rank Sum Test			A measure
		Rank Sum	p value	Significant	
2	Sum-of-N	2297	0.6328	No	–
	Factorial	2198	0.3183	No	–
	Fibonacci	2821	0.0619	No	–
	Binary-Sum	3179	0.4178	No	–
	Reverse	2079	0.2116	No	–
4	Sum-of-N	2139	0.0926	No	–
	Factorial	2851	0.0857	No	–
	Fibonacci	3298	0.0618	No	–
	Binary-Sum	2711	0.5161	No	–
	Reverse	2628	0.5119	No	–

Cores	Program	Wilcoxon Signed Rank Sum Test			A measure
		Rank Sum	p value	Significant	
8	Sum-of-N	2189	0.0432	Yes	0.4183
	Factorial	2815	0.0408	Yes	0.4917
	Fibonacci	3321	0.0018	Yes	0.6313
	Binary-Sum	2079	0.0032	Yes	0.7851
	Reverse	2479	0.0481	Yes	0.4819
16	Sum-of-N	2250	0.0481	Yes	0.4041
	Factorial	2701	0.0097	Yes	0.5151
	Fibonacci	3253	0.0061	Yes	0.6559
	Binary-Sum	2221	0.0057	Yes	0.6956
	Reverse	2008	0.0174	Yes	0.5051

Fig. 5: Significance tests (at $\alpha = 0.05$) show that ALP is competitive with MCGE-II on all the cores except 2. "Yes" states the results are significant and "No" states that they are insignificant. A measure shows the probability at which, ALP is better over the other.

cores) with that of the standard GE generated serial programs for all the five experimental benchmark programs. Note, the serial programs are generated automatically by omitting the OpenMP parallelization pragmas from the grammar shown in Fig. 2. It is important to compare the performance of parallel programs with that of the serial counterparts, as poorly crafted parallel programs often obviate to achieve the promised gains. That being the fact, it is always interesting when the evolving parallel programs achieve lock-freedom. The difference between the AET results of lock-free and the serial programs is insignificant when they are scheduled to execute on 1-core of the processor.

Fig. 3 shows the Wilcoxon Signed Rank Sum tests (at a significance level of $\alpha = 0.05$) between ALP and the serial counterpart at varying number of cores. As expected, ALP significantly outperforms the standard GE when the resultant programs are scheduled to execute on 4, 8, and 16 cores. But, surprisingly, there is an insignificant difference between the two methods for 2 cores. In other words, a slight increase in the processing power has negligible effect on the performance of the ALP resultant programs. It is because of the extra overhead incurred in scheduling the threads, which was originally manifested in [6], thus the performance fails to improve for 2 cores. Fig. 3 also shows Vargha-Delaney *A*-measure [27] that describes the probability at which, ALP is better than its serial counterpart. This measure is applicable only on the results that are proved to be statistically significant. In essence: when the `A-measure > 0.5` then ALP is better than standard GE; when the `A-measure < 0.5` then ALP is worse than the other; when `A-measure = 0.5` then both the methods are equal. The greater the magnitude of `A-measure`, better is the performance difference.

Overall, auto-generated parallel recursive lock-free programs exhibit a significantly better performance (in terms of execution time) over their serial counterparts. Particularly, the synthesized lock-free programs automate the use of *atomic* updates (*fetch-and-add* operations) on shared variables. Next, we compare the speed-up of the ALP synthesized parallel lock-free programs with that of the MCGE-II synthesized parallel programs.

2) Speed-up: Speed-up is the ratio of the average execution times of an approach on 1 core (AET_{1-core}) to n cores ($AET_{n-cores}$), which is shown in Eq. 3. We compare the speed-up of ALP resultant programs with that of the MCGE-II. Fig. 4 shows the speed-up of both the approaches for all the five programs at varying number of cores (2, 4, 8, and 16).

$$\text{Speed-up} = \frac{AET_{1-core}}{AET_{n-cores}} \tag{3}$$

Fig. 5 presents the Wilcoxon Signed Rank Sum significance tests (at $\alpha = 0.05$) between ALP and MCGE-II with *A*-measure. The results indicate that there is no difference between ALP and MCGE-II when the respective resultant programs are scheduled to execute on 2 and 4 cores. On the other hand, ALP significantly outperforms MCGE-II on 8 and 16 cores. That is, the impact of lock-free programs is better observed with an increase in the number of cores. Overall for 16 cores, ALP results indicate an average (on all problems) speed-up of 10.43, a significantly better improvement of 25.21% over MCGE-II, which reports an average speed-up of 8.33. However, an analysis on the *efficiency* of these programs better explains the impact of lock-free programming.

3) Efficiency: Efficiency of an approach is the ratio of speed-up over the number of cores and lies in between 0 and 1. It is often difficult to achieve ideal (1) efficiency, nevertheless useful in analysing the scale-up of both the methods.

TABLE IV: Efficiency of ALP and MCGE-II.

Program	ALP				MCGE-II			
	Cores				Cores			
	2	4	8	16	2	4	8	16
1	0.57	0.70	0.62	0.65	0.61	0.64	0.53	0.47
2	0.46	0.74	0.68	0.71	0.55	0.65	0.54	0.44
3	0.49	0.79	0.76	0.71	0.51	0.74	0.60	0.53
4	0.74	0.74	0.86	0.78	0.78	0.77	0.63	0.57
5	0.48	0.65	0.80	0.76	0.51	0.73	0.61	0.56

Table IV presents the efficiency of ALP and MCGE-II for all five benchmark programs at different cores. It is evident from the results that the efficiency of both the methods on 2 and 4 cores is similar, whereas ALP is better than MCGE-II for 8 and 16 cores, an improvement in the performance at higher number of cores with lock-free programs. That shows the ALP synthesized parallel lock-free programs exhibit better performance over the MCGE-II generated parallel programs. Next, the lessons learnt from the experimental evaluation are discussed.

V. Discussion

We discuss the intelligence as well as the competence of the resultant lock-free programs. The intelligence is justified by focusing on the correctness of the resultant programs, while the competence is measured by comparing their performance with that of the human written parallel programs.

A. Best Program

Fig. 6 shows the ALP generated program that uses `#pragma omp parallel sections` directive. The result (`res`) is shared among the threads, while the variable `n` is thread-private. The recursive calls are computed independently in parallel, where `res` is updated atomically by using the GCC atomic `__sync_add_and_fetch` primitive. On executing this primitive, `res` gets a new value, thus avoids the locking problem. Next, we discuss the correctness of the synthesized programs.

1) Correctness: Since evolution searches through the compilable solution space, all the solutions except the best, become sub-optimal solutions. For example, line 14 of Fig. 6 contains `__sync_add_and_fetch`, a sub-optimal solution of this is `__sync_sub_and_fetch`, which in fact is an incorrect solution.

```
1int fib(int n) {
  int a;
3 if (n <= 2) {
   int a=n;
5  __sync_fetch_and_add(res, a);
  }
7 else {
   #pragma omp parallel sections \
9  shared(res) private(n)
   { //more than one thread in operation
11   #pragma omp section
     {
13     int a = fib(n-1);
       __sync_add_and_fetch(res, a);
15   }
     #pragma omp section
17   {
       int a = fib(n-2);
19     __sync_add_and_fetch(res, a);
     }
21 } } return res; }
```

Fig. 6: ALP synthesized best-of-run parallel *lock-free* recursive Fibonacci program.

⟨*lockfree*⟩ ::= `__sync_fetch_and_add` (0)
| `__sync_fetch_and_sub` (1)
| `__sync_add_and_fetch` (2)
| `__sync_sub_and_fetch` (3)

Fig. 7: Production rules of the *lockfree* non-terminal.

Being a heuristic technique, ALP can generate sub-optimal programs. Notice the non-terminal `<lock-free>` shown in Fig. 7, that contains four production rules for atomic updates. For Fibonacci program, rule (2) forms part of an optimal solution while the programs with the remaining three rules are sub-optimal. The best programs in the earlier generations of the evolutionary cycle often contain such sub-optimal solutions. Although these programs are incorrect, their execution time is considered in fitness evaluation to improve the overall performance and for a fair comparison with standard GE and MCGE-II.

On the contrary, the non-terminal `<omppragma>` in Fig. 2 might select a wrong parallelization pragma. In those situations, the respective individual gets a zero fitness as explained in section III-A. Overall, ALP generates an optimal solution (Fig. 6) in all the evolutionary runs for all the programs. Next, we discuss the human competitive nature of the synthesized parallel lock-free recursive programs.

B. Human Competitive

While it is obvious for a human programmer to employ the `critical directive` to ensure correctness of the program, evolution in GE entails a learning curve where it learns about program semantics as well as the most suitable parallel constructs. Therefore, this paper presents a valuable proof of concept even if the target programs have obvious human programmed solutions. Our further work will explore applications where forgoing locks adds significant programming complexity and/or when the program clearly requires performance optimisation and can benefit from an optimisation algorithm such as GE.

TABLE V: Efficiency of the best programs from the evolutionary process versus human developed programs.

#	ALP Cores		Human Cores		Wilcoxon *p*-value		A measure	
	8	16	8	16	8	16	8	16
1	0.78	0.59	0.81	0.67	0.57	0.09	–	–
2	0.54	0.61	0.64	0.65	0.10	0.60	–	–
3	0.76	0.63	0.60	0.54	**0.03**	**0.04**	0.47	0.32
4	0.81	0.75	0.68	0.62	**0.01**	**0.03**	0.59	0.38
5	0.73	0.58	0.69	0.61	0.07	0.1	–	–

We compare the efficiency of the best evolved programs (Ex:- Fig. 6) with that of the human written programs to assess the relative merits and demerits of ALP. Note, the human written program is same as the program shown in Fig. 1 except the fact that they use `#pragma omp atomic` instead of the `critical` construct.

Table V compares the efficiency of all the five best-of-run parallel lock-free recursive programs with that of the human written programs on 8 and 16 cores. It presents the Wilcoxon Signed Rank Sum tests with Vargha-Delaney *A*-measure. The *p*-value is in boldface for the corresponding program when the efficiency of ALP programs is significantly better than that of the human written programs. Overall, the results indicate that the automatically generated lock-free programs are competitive with that of the human written programs. However, in both the methods the programs are limited by the Linux kernel scalability issues in reaching from ideal efficiency.

The auto-generated lock-free programs compete with that of the human written programs. It is due to the fact that the latter uses OpenMP *atomic* that is native to that library, hence does not require any extra overhead in calling this built-in function as opposed to a system call for the GCC atomic memory calls. However, OpenMP atomic can deal only with small operations. In case of larger operations it fails to guarantee same scale of performance, whereas ALP resultant programs do not pose such limitations. However, GCC atomic primitives pose a few architectural limitations.

C. Limitations

We list some of the limitations of ALP. As explained in section V-A use of an inapt atomic primitive results in the generation of poor solutions. Quite rarely, use of a correct primitive may still fail to produce a correct solution. When the target CPU architectures does not support 16-byte types, then there is no guarantee for the atomicity of `read-modify-write` operations. The CPU architecture used in this paper does not support these types. Thus, it is

necessary to develop novel lock based synchronization constructs (as explain in [1]) in order to improve the scalability of lock-free algorithms on multi-cores.

VI. Conclusions and Future Work

We presented ALP, a genetic improvement approach, which synthesized parallel lock-free recursive programs. The automatic lock-free programs used both OpenMP parallelization pragmas and GNU GCC atomic primitives. The resultant programs have efficiently exploited the compute capability of the multi-core processors. These programs proved to significantly outperform the evolved serial and the parallel programs at higher number of cores. Along with that, the performance of these programs is competitive with that of the human written parallel programs.

This avenue of research can be extended in a number of directions. First, we can automate the memory reordering operations. Next, we intend to address the thin line difference between the wait-free synchronization [13] and the lock-free programming through program synthesis. Third, ALP fitness evaluation (Eq. 1) has twin objectives of achieving program correctness and execution time. Thus, we recommend to explore both the hardness of test cases (for example, lexicase selection [12]) and the structured approaches used in behavioural program synthesis [14]. Finally, we intend to investigate real-world problems in order to limit the gap between theory and application.

References

[1] S. Al Bahra. Nonblocking algorithms and scalable multicore programming. *Queue*, 11(5):40:40–40:64, 2013.

[2] J. D. Álvarez, J. M. Colmenar, J. L. Risco-Martín, J. Lanchares, and O. Garnica. Optimizing l1 cache for embedded systems through grammatical evolution. *Soft Computing*, 2015.

[3] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of IEEE Congress on Evolutionary Computation*, pages 162–168, 2008.

[4] G. Chennupati, R. M. A. Azad, and C. Ryan. On the automatic generation of efficient parallel iterative sorting algorithms. In S. Silva and A. I. Esparcia-Alcázar, editors, *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, 2015.

[5] G. Chennupati, R. M. A. Azad, and C. Ryan. Performance optimization of multi-core grammatical evolution generated parallel recursive programs. In S. Silva and A. I. Esparcia-Alcázar, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1007–1014. ACM, 2015.

[6] G. Chennupati, J. Fitzgerald, and C. Ryan. On the efficiency of multi-core grammatical evolution (MCGE) evolving multi-core parallel programs. In *Proceedings of the 6th World Congress on Nature and Biologically Inspired Computing*, pages 238–243, 2014.

[7] B. Cody-Kenny, E. G. Lopez, and S. Barrett. locoGP: improving performance by genetic programming java source code. In *Proceedings of Genetic and Evolutionary Computation Conference companion*, pages 811–818. ACM, 2015.

[8] D. Dechev, P. Laborde, and S. Feldman. Lc/dc: Lockless containers and data concurrency a novel nonblocking container library for multicore applications. *IEEE Access*, 1:625–645, 2013.

[9] M. Desnoyers and M. R. Dagenais. Lockless multi-core high-throughput buffering scheme for kernel tracing. *SIGOPS Operating Systems Review*, 46(3):65–81, 2012.

[10] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–14. ACM, 2012.

[11] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of 15th International Conference on Distributed Computing*, pages 300–314. Springer, 2001.

[12] T. Helmuth, L. Spector, and J. Matheson. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation*, 19(5):630–643, 2015.

[13] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[14] K. Krawiec, J. Swan, and U.-M. O'Reilly. Behavioral program synthesis: Insights and prospects. In *Genetic Programming Theory and Practice*. Springer, 2015.

[15] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[16] W. B. Langdon. Genetic improvement of software for multiple objectives. In Y. Labiche and M. Barros, editors, *SSBSE*, volume 9275 of *LNCS*, pages 12–28. Springer, 2015.

[17] W. B. Langdon. Performance of genetic programming optimised Bowtie2 on genome comparison and analytic testing (GCAT) benchmarks. *BioData Mining*, 8(1), 2015.

[18] W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In P. Sobrevilla, editor, *IEEE World Congress on Computational Intelligence*, pages 2376–2383. IEEE, 2010.

[19] W. B. Langdon and M. Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, 2015.

[20] M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[21] M. Orlov and M. Sipper. Flight of the finch through the java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, 2011.

[22] 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, *Proceedings of the Second Annual Conference on Genetic Programming*, pages 295–302. Springer, 1997.

[23] W. R. Stevens and S. A. Rago. *Advanced Programming in the UNIX Environment*. Addison-Wesley Professional, 3rd edition, 2013.

[24] S. Timnat and E. Petrank. A practical wait-free simulation for lock-free data structures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 357–368. ACM, 2014.

[25] F. Tordini, M. Aldinucci, and M. Torquati. High-level lock-less programming for multicore. In *Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES) – Poster Abstracts*, 2012.

[26] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222. ACM, 1995.

[27] A. Vargha and H. D. Delaney. A critique and improvement of the "cl" common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[28] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st IEEE International Conference on Software Engineering*, ICSE 2009, pages 364–374, 2009.

[29] D. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, 2011.