Performance Optimization of Multi-Core Grammatical Evolution Generated Parallel Recursive 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

Although Evolutionary Computation (EC) has been used with considerable success to evolve computer programs, the majority of this work has targeted the production of serial code. Recent work with Grammatical Evolution (GE) produced *Multi-core Grammatical Evolution (MCGE-II)*, a system that *natively* produces parallel code, including the ability to execute recursive calls in parallel.

This paper extends this work by including practical constraints into the grammars and fitness functions, such as increased control over the level of parallelism for each individual. These changes execute the best-of-generation programs faster than the original MCGE-II with an average factor of 8.13 across a selection of hard problems from the literature.

We analyze the time complexity of these programs and identify avoiding *excessive parallelism* as a key for further performance scaling. We amend the grammars to evolve a mix of serial and parallel code, which spawns only as many threads as is efficient given the underlying OS and hardware; this speeds up execution by a factor of 9.97.

Categories and Subject Descriptors

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

Keywords

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

1. INTRODUCTION

Multi-core processors are shared memory multiprocessors integrated on a single chip and offer high processing power. However, some challenges remain in exploiting their potential, particularly because, as the number of cores increase (the so-called *death of scaling*[1]), the software needs to be designed explicitly to realize the true potential of these cores.

[1]http://www.gotw.ca/publications/concurrency-ddj.htm

GECCO '15, July 11 - 16, 2015, Madrid, Spain

DOI: http://dx.doi.org/10.1145/2739480.2754746

OpenMP [13] is the *de facto* standard to write parallel programs on multi-cores in C/C++. Although a powerful tool, writing that is a non-trivial task, as sequential programmers still struggle with parallel design issues such as synchronization, locking and program decomposition.

To alleviate this difficulty, [6] introduced *MCGE-II* to generate *native* parallel code on multi-cores, proved it by parallelising recursive calls. MCGE-II adds OpenMP pragmas in GE grammars, and so solves the underlying problem such that the solution is a parallel program. Although successful, it made little attempt to optimize the efficiency, particularly in terms of the execution time and the ease of program evolution.

We re-design the MCGE-II grammars such that they now partition *data and task level parallelism* under different production rules making it convenient for evolution to select as appropriate. Furthermore, we modify the fitness function to explicitly take account of the execution time of the individuals. These changes combine to give an average speed-up factor of 8.13 across a range of hard recursive problems from the literature. Recursion is perfect for task level parallelism as each recursive call can be invoked in a separate thread.

However, one risk in evolving parallel recursion is *excessive* multi-threading that spawns a thread at every recursive call; thus, each thread does little useful work. This can degrade performance due to the overhead in first creating and then scheduling these threads. Scheduling can be especially expensive as a large number of threads compete to get a slice of CPU time. Therefore, we further tweak the grammars such that the evolved programs can control the creation of threads after a certain depth in their recursive trace. Thus, the lower level recursive calls run in serial by arresting thread creation, while the top level calls run in parallel, that results in an average speed-up of 9.97.

We also consider code growth in GE but find it surprisingly insignificant; therefore, code-growth does not affect program execution.

The rest of the paper is detailed as follows: section 2 introduces the existing literature of the paper; section 3 describes the proposed approach; section 4 presents the experiments; section 5 shows the results; section 6 discusses the factors that influence the performance, while section 7 enhances the performance; and finally, section 8 concludes.

2. BACKGROUND

Unlike MCGE-II, previous EC work has only treated evolving recursion and evolving parallel programs as separate challenges. Below we only briefly review these two topics.

2.1 Generation of Recursive Programs

Some of the earliest work on evolving recursion is from Koza [9, Chapter-18] which evolved a Fibonacci sequence; this work cached previously computed recursive calls for efficiency. Also, Brave [4] used *Automatically Defined Functions* (ADFs) to evolve recursive tree search. In this, recursion terminated upon reaching the tree depth. Then, [17] concluded that infinite recursions was a major obstacle to evolve recursive programs. However, [19] successfully used an adaptive grammar to evolve recursive programs; the grammar adjusted the production rule weights in evolving solutions.

Spector et al. [15] evolved recursive programs using PushGP by explicitly manipulating its execution stack. The evolved programs were of $\mathrm{O}(n^2)$ complexity, which became $\mathrm{O}(nlog(n))$ with an efficiency component in fitness evaluation.

Agapitos and Lucas [1, 2] evolved recursive quick sort with an Object Oriented Genetic Programming (OOGP) in Java. The evolved programs exhibited a time complexity of $\mathrm{O}(nlog(n))$. Recently, Moraglio et al. [10] used a non-recursive scaffolding method to evolve recursive programs with a context free grammar based GP.

Next, we review automatic parallel programming with EC.

2.2 Automatic Evolution of Parallel Programs

This section describes research into the automatic generation of parallel programs irrespective of recursion. In general, automatic generation of parallel programs can be divided into two types: *auto-parallelization of serial code* and the *generation of native parallel code.*

Auto-parallelization requires an existing (serial) program. Using GP, [14, Chapter-5] proposed *Paragen* which had initial success, however, its reliance on the execution of candidate solutions ran into difficulties with complex and time consuming loops. Later, *Paragen-II* [14, Chapter-7] dealt the loop inter-dependencies, relying on a rough estimate of execution time. Then, [14] extended *Paragen-II* to merge independent tasks of different loops into one loop.

Similarly, genetic algorithms evolved *transformations*; [11] and [18] proposed *GAPS* (Genetic Algorithm Parallelization System) and, *Revolver* respectively. GAPS evolved sequence restructuring, while *Revolver* transformed the loops and programs, both optimized the execution time.

On the other hand, *native parallel code generation* produces a working program that is also parallel. With multi-tree GP, [16] concurrently executed autonomous agents for automatic design of controllers.

Recently, Chennupati et al., [5] evolved natively parallel regression programs. Then [6] introduced MCGE-II to evolve task parallel recursive programs. The minimal execution time of them was merely due to the presence of OpenMP pragmas which automatically map threads to cores. However, the use of a different OpenMP pragma alters the performance of the parallel program, and skilled parallel programmers carefully choose the pragmas when writing code. To that end, in this paper, we extend MCGE-II in two ways: we re-structure the grammars so task and data level parallelism is separate, and we explicitly penalize long executions.

3. MCGE-II

In this paper we extend MCGE-II [6] in two respects: first, through the design of the grammars (section 3.1), which are much richer and categorize rules better so as to accelerate

⟨*program*⟩	::=	⟨*condition*⟩ ⟨*parcode*⟩
⟨*omppragma*⟩	::=	⟨*ompdata*⟩ | ⟨*omptask*⟩
⟨*ompdata*⟩	::=	#pragma omp parallel
	|	#pragma omp master
	|	#pragma omp single
	|	#pragma omp parallel for
⟨*omptask*⟩	::=	#pragma omp parallel sections
	|	#pragma omp task
⟨*shared*⟩	::=	shared(⟨*input*⟩, temp, res) ⟨*newline*⟩ '{'
⟨*private*⟩	::=	private(a) | firstprivate(a) | lastprivate(a)
⟨*condition*⟩	::=	if(⟨*input*⟩ ⟨*lop*⟩ ⟨*const*⟩) '{' ⟨*newline*⟩ ⟨*line1*⟩; ⟨*newline*⟩ ⟨*line2*⟩; ⟨*newline*⟩'}'
⟨*parcode*⟩	::=	else '{' ⟨*newline*⟩ ⟨*omppragma*⟩ ⟨*private*⟩ ⟨*shared*⟩ ⟨*blocks*⟩ ⟨*newline*⟩ '}' ⟨*newline*⟩ '}' ⟨*newline*⟩ ⟨*result*⟩
⟨*blocks*⟩	::=	⟨*parblocks*⟩ | ⟨*blocks*⟩ ⟨*newline*⟩ ⟨*blocks*⟩
⟨*parblocks*⟩	::=	⟨*secblocks*⟩ | ⟨*taskblocks*⟩
⟨*secblocks*⟩	::=	#pragma omp section ⟨*newline*⟩ '{' ⟨*newline*⟩ ⟨*line1*⟩; ⟨*newline*⟩ ⟨*atomic*⟩ ⟨*newline*⟩ ⟨*line2*⟩ ⟨*bop*⟩ a; ⟨*newline*⟩'}'
⟨*taskblocks*⟩	::=	#pragma omp task ⟨*newline*⟩ '{' ⟨*newline*⟩ ⟨*line1*⟩; ⟨*newline*⟩ ⟨*atomic*⟩ ⟨*line2*⟩ ⟨*bop*⟩ a;⟨*newline*⟩'}'
⟨*atomic*⟩	::=	#pragma omp atomic
⟨*line1*⟩	::=	temp = ⟨*expr*⟩ | a = ⟨*expr*⟩;
⟨*line2*⟩	::=	res ⟨*bop*⟩= temp
⟨*expr*⟩	::=	⟨*input*⟩ | ⟨*stmt*⟩ | ⟨*stmt*⟩ ⟨*bop*⟩ ⟨*stmt*⟩
⟨*result*⟩	::=	return res;
⟨*stmt*⟩	::=	fib(⟨*input*⟩ ⟨*bop*⟩ ⟨*const*⟩);
⟨*input*⟩	::=	n
⟨*lop*⟩	::=	'>=' | '<=' | '>' | '<' | '=='
⟨*bop*⟩	::=	+ | - | * | /
⟨*const*⟩	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨*newline*⟩	::=	$\backslash n$

Figure 1: The design of MCGE-II grammar to generate natively parallel recursive Fibonacci programs.

evolution; and second, through a modified fitness function (section 3.2), which considers time in its evaluation. Both of them help optimize the performance (execution time) of the evolving parallel recursive programs.

3.1 Design of Grammars

The grammars in this paper are designed such that they offer clear separation among OpenMP pragmas, task and data parallel. This separation benefits the quick generation of candidate solutions because of grammatical bias [17]. Figure 1 shows the grammar that generates a natively parallel recursive *Fibonacci* program. The non-terminals `<omptask>` and `<ompdata>` represent the task and data parallel pragmas respectively, while `<omppragma>` selects one of the two options. Notice, the generation of the task level pragma shown in `<omptask>` forms the best fit individual as the goal is the automatic generation of task parallel recursion.

The programs take an integer (n) input (`<input>`), while the variable *res* returns the end result of the parallel program evaluation. Moreover, the two local variables (*temp, a*) store the intermediate results of recursive calls. The input and the two variables are shared among the threads with the clause (`<shared>`), while `a` is a thread private (`<private>`) variable. Evolution selects a private clause from the three OpenMP private (`<private>`) clauses.

Of the three private clauses: *private(a)* makes a variable thread specific, any changes on the variable are invisible after the parallel region; *firstprivate(a)* holds a value throughout the program despite the parallelization; *lastprivate(a)* keeps the changes of the last thread in the parallel region. Since the variable updates are thread specific, programs with *private(a)* are the best programs. Note, the other clauses degrade the fitness as they evolve incorrect solutions.

The non-terminal symbol `<stmt>` depicts the recursive call of the Fibonacci program. The non-terminal `<blocks>` generates a sequence of parallel blocks with each block containing an independent recursive call. To that end, section 7 presents a complete task parallel recursive program.

3.2 Performance Optimization

As the choice of OpenMP pragmas can significantly impact the performance of a program, we encourage the right degree of parallelism with its execution time in the fitness, which increases the pressure in choosing the best pragma.

Thus, the fitness function that we use in optimizing the performance is the product of two factors: execution time and the mean absolute error, both are normalized in the range $(0, 1)$ – a maximization function. The following equation represents the fitness of the evolved program (f_{pprog}):

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

where, t is the time taken by the evolved parallel program to evaluate across all the training cases (N); The terms y_i and, $\hat{y}_i$ represent the actual and, the evolved outputs respectively.

Since using an inapt pragma increases the time to execute the evolved program, the first term, *normalized execution time* in eq. 1 helps to select the correct pragma. Meanwhile, the second term, *normalized mean absolute error* (in eq. 1) enforces program correctness. Together, the two objectives push for a correct and efficient parallel program.

Table 1: The summary of the problems (in increasing order of difficulty) under investigation with the properties used in the experiments.

#	Problem	Type		LV	Range
		Input	Return		
1	Sum-of-N	int	int	3	[1,1000]
2	Factorial	int	unsigned long long	3	[1,60]
3	Fibonacci	int	unsigned long long	3	[1,60]
4	Binary-Sum	int [ ], int, int	int	2	[1,1000]
5	Reverse	int [ ], int, int	void	2	[1,1000]
6	Quicksort	int [ ], int, int	void	3	[1,1000]

4. EXPERIMENTAL CONTEXT

We evaluate our approach on six recursive problems. Table 1 presents them with their properties: type of input and return values, the number of arguments, number of local variables (LV); the range of elements from which the input is considered. The output is the result of the conventional algorithm of the respective problem. The local variables (LV) are the auxiliary variables. The training set contains 30 data points. The first three (Sum-of-N, Factorial, Fibonacci) problems accept a single positive integer as input; for *Sum-of-N*, it is randomly generated from the range $[1, 1000]$ while, for *Factorial* and *Fibonacci* problems, it is in the range $[1, 60]$ due to the limitations in the data type range in C. While the remaining three problems (Binary-Sum, Reverse, Quicksort) accept an array of integers along with their start and end indices as input, for which, an array of 1000 elements are randomly generated from the range $[1, 1000]$. Note, the grammars are general enough except for a few minor changes with respect to the problem at hand.

Table 2 describes the algorithmic parameters along with the experimental environment used to evaluate our approach.

Table 2: Parameters and experimental environment.

GE parameter settings	
Parameter	Value
point mutation	0.1
one point crossover	0.9
selection	Roulette Wheel
replacement strategy	Steady state
initialization	Sensible
minimum depth	9
maximum depth	25
wrapping	disabled
population size	500
generations	100
runs	50
Experimental environment	
CPU	Intel (R) Xeon (R) E7-4820, 16 cores
OS	Debian Linux v 2.6.32, 64-bit
C++	GNU GCC v 4.4.5 libGE v 0.26
OpenMP	libgomp v 3.0
Timer utility	*omp_get_wtime*()

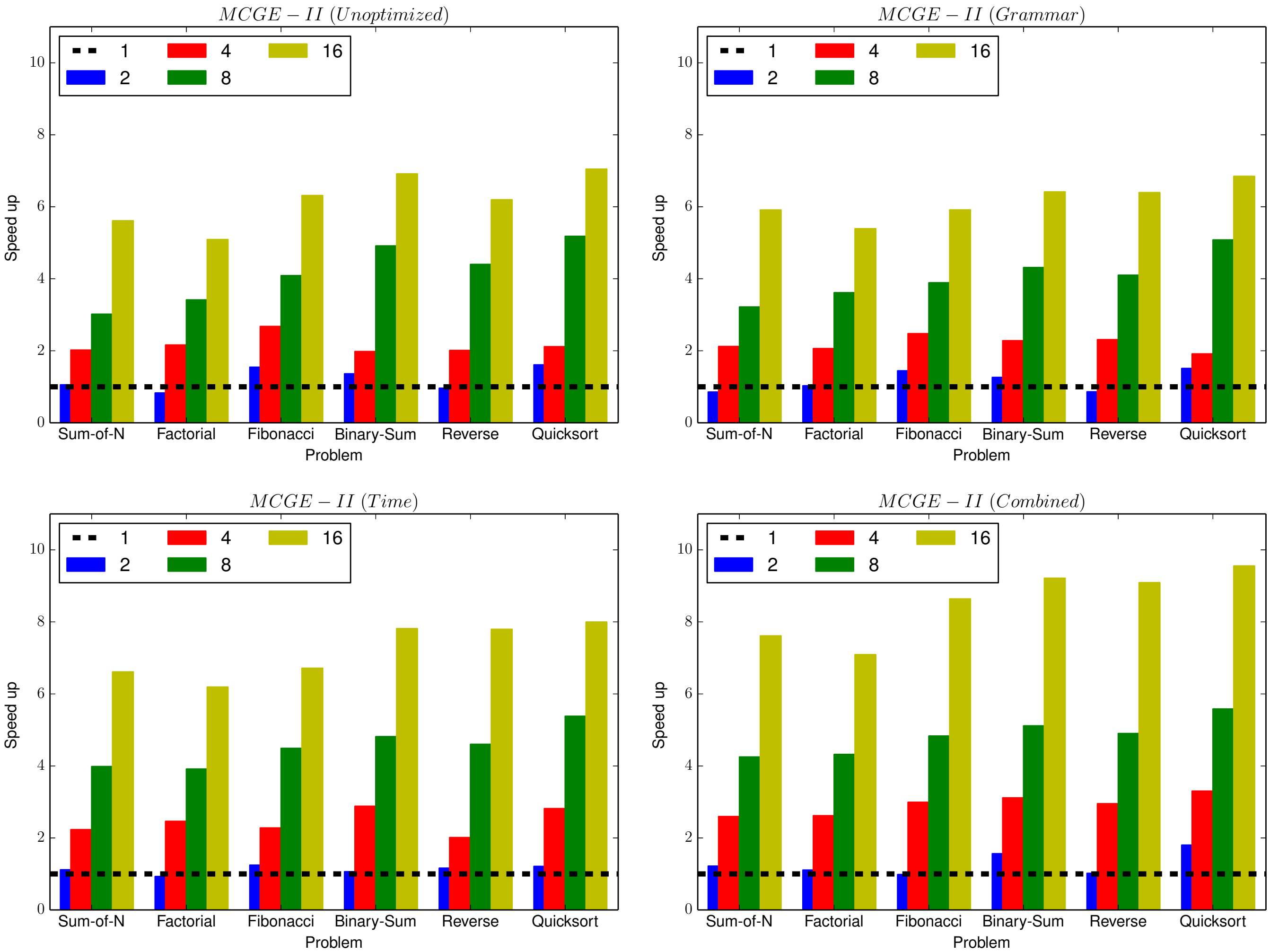

Figure 2: The speed-up of *MCGE-II (Unoptimized, Grammar, Time, Combined)* variants for all the six experimental problems. The number of cores vary as 2, 4, 8 and 16. The horizontal dashed line (–) represents the speed-up of 1 and acts as a reference for the remaining results.

Table 3: Friedman statistical tests with Hommel's post-hoc analysis on performance of all the four MCGE-II variants. The boldface shows the significance at $\alpha = 0.05$, while asterisk (*) shows the best variant.

Cores	MCGE-II variant	Average Rank	p value	p Hommel
4	Unoptimized	**3.25**	**0.0025**	**0.0167**
	Grammar	**3.1667**	**0.0036**	**0.025**
	Time	**2.5833**	**0.0331**	**0.05**
	Combined*	0.9999	-	-
8	Unoptimized	**3.1667**	**0.0036**	**0.025**
	Grammar	**3.6667**	**3.47E-4**	**0.0167**
	Time	2.1667	0.1175	0.1
	Combined*	1.0	-	-
16	Unoptimized	**3.4999**	**7.96E-4**	**0.0167**
	Grammar	**3.4999**	**7.96E-4**	**0.025**
	Time	1.9999	0.1797	0.05
	Combined*	1.0	-	-

Table 4: The mean best generation (mean ± [standard deviation]) of all the four MCGE-II variants (Unoptimized, Grammar, Time, Combined). The lowest value is in boldface.

#	MCGE-II			
	Unoptimized	Grammar	Time	Combined
	mean best generation	mean best generation	mean best generation	mean best generation
1	59.14 ± [4.96]	45.38 ± [2.81]	51.63 ± [6.19]	**43.27** ± **[5.37]**
2	38.43 ± [2.85]	**31.19** ± **[4.73]**	39.35 ± [3.19]	36.51 ± [3.67]
3	77.36 ± [5.58]	**44.73** ± **[5.26]**	65.19 ± [6.43]	59.89 ±[4.15]
4	71.83 ± [6.37]	**59.14** ± **[5.34]**	68.88 ± [4.51]	61.43 ± [5.19]
5	56.68 ± [2.19]	47.53 ± [2.19]	51.09 ± [2.39]	**45.32** ± **[4.92]**
6	49.25 ± [4.57]	**40.49** ± **[5.23]**	52.49 ± [2.58]	47.28 ± [3.15]

Friedman tests with Hommel's post-hoc analysis. Boldface shows the significance at $\alpha = 0.05$, while asterisk (*) shows the best variant.

MCGE-II variant	Average Rank	p-value	p-Hommel
Unoptimized	**3.0**	**5.3205E-4**	**0.001596**
Grammar*	1.16666	-	-
Time	**2.33333**	**0.0321438**	**0.042**
Combined	**1.99999**	**0.0024787**	**0.02**

We compare the performance among four MCGE-II variants. The first variant, named as *MCGE-II (Unoptimized)* hereafter, does not separate task and data parallelism; instead, all the rules in `<omptask>` and `<ompdata>` are lumped together under `<omppragma>`. Thus, it is hard to omit data parallelism when it only requires task parallelism, while the fitness function is *only* the *normalized mean absolute error* (second term in eq. 1). The second variant (*MCGE-II (Grammar)*), uses the grammars shown in Figure 1, while the fitness function is the *normalized mean absolute error*. The third variant (*MCGE-II (Time)*), uses the same grammars as with the first variant (MCGE-II (Unoptimized)), but f_{pprog} (eq. 1) for fitness evaluation. The fourth variant, named as *MCGE-II (Combined)* hereafter, uses both the grammars as in Figure 1 and the fitness function f_{pprog}.

5. EXPERIMENTAL RESULTS

In this section we report the speed-up in terms of the *mean best execution time (MBT)*, that is, the average execution time of the best of generation programs in each of 50 runs. Now, the Speed-up $= \frac{T_{MBT-1-core}}{T_{MBT-n-cores}}$ where, $T_{MBT-1-core}$ is the *mean best execution time* on a single core, while $T_{MBT-n-cores}$ is that of n-cores of a processor.

Figure 2 presents the speed-up of all the four MCGE-II (*Unoptimized, Grammar, Time, Combined*) variants across all the six experimental problems. The speed-up is for varying the cores: 2, 4, 8, and 16, with respect to 1-core.

Table 3 shows the Friedman statistical tests with Hommel's post-hoc [8] analysis on the speed-up of the four MCGE-II variants for all the six problems at $\alpha = 0.05$. The first column indicates the number of cores under execution. The second column shows the MCGE-II variant, while the third column presents the average rank. The fourth and the fifth columns show the p-value and Hommel's critical value respectively. The lowest average rank shows the best (MCGE-II (Combined)) variant, and is marked with an asterisk (*). A variant is significantly different from the best variant if p-value is less than p-Hommel at $\alpha = 0.05$, is in boldface.

Although we do not show in Table 3, for 2 cores none of the variants is better than the others, which could be attributed to the multi-threading overhead offsetting the performance gains. However, we see differences with higher numbers of cores. For 4 cores, MCGE-II (Combined) significantly outperforms the remaining three variants (Unoptimized, Grammar, Time), while for 8 and 16 cores MCGE-II (Combined) significantly outperforms the two MCGE-II (Unoptimized, Grammar) variants, but the difference with MCGE-II (Time) is insignificant. We believe that this is due to the fact that both in MCGE-II (Time, Combined), include execution time in fitness evaluation.

We also investigate the effect of restructuring grammars in this study, on the ease of evolving the correct programs. To this end, we measure the mean number of generations required to converge to the best fitness, with a pre-condition that the program under consideration must be *correct*, averaged across 50 runs; we call it the *mean best generation*. Table 4 shows the mean best generation of all the four variants[2] along with significance tests. The lowest mean best generation is in boldface for the respective problem of an MCGE-II variant. The significance tests show that MCGE-II (Grammar) significantly outperforms MCGE-II (Unoptimized, Time, Combined) at $\alpha = 0.05$. That is MCGE-II (Grammar) requires less number of generations over the other variants in producing the best fit programs due to the grammatical bias [17] exerted through the changes (Figure 1) in the design of grammars.

However, the performance results (Table 3) indicate that, although MCGE-II (Grammar) quickly generates the parallel recursive programs, it has been outperformed by MCGE-II (Time, Combine) in terms of their efficiency. A pairwise comparison between MCGE-II (Time, Combined) at $\alpha = 0.05$ shows that MCGE-II (Combined) outperforms MCGE-II (Time) in terms of number of generations. That is, MCGE-II (Combined) quickly generates efficient parallel recursive programs than MCGE-II (Time) due to the grammatical bias. Hence, MCGE-II (Combined) is the best variant that reports an average (on all the problems) speed-up of 8.13 for 16 cores, a significant improvement of 23.86% over MCGE-II (Unoptimized) that reports 6.19 speed-up.

Although MCGE-II (Combined) boosts up the performance over the original system, it fails to fully utilize the power of multi-cores. That is, it reported a speed-up of 8.13 on 16 cores of processor, where the ideal case should be 16. Many factors contribute in obviating to achieve the ideal scale-up.

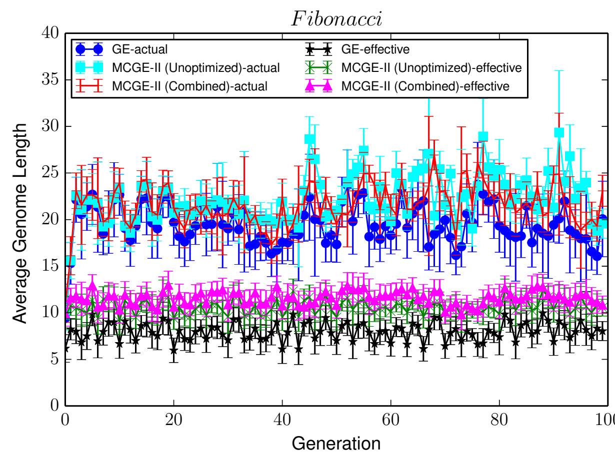

Figure 3: The lengths of GE, MCGE-II (Unoptimized, Combined) genotypes for *Fibonacci*. Space constraints do not allow reporting the graphs for all the problems but the trends are very similar.

6. DISCUSSION

The quality of parallel code is difficult to quantify as execution time often depends on the ability of OS to efficiently schedule the tasks. This job itself is complicated by other parallel threads (from other programs) running at the same time. OpenMP abstracts away much of these concerns from programmers, which makes it easier at the cost of some of fine control. In order to compensate this, it is often necessary to dig deeper by adapting to program to the hardware, which makes the programmer's job increasingly hard.

Hardware can cap the maximum number of threads; however, in the given grammars each recursive call spawns a new thread. Then, the OS-specific factors for the Linux ker-

[2]Note, the results are for 16 cores. They are similar for the remaining cores, but left out due to space constraints.

nels, which eventually fail to scale in scheduling the very high number of threads. Moreover, a parent thread spawns a child thread, it sleeps until all the child threads have finished. This process expensive, when a large number of threads are involved. Also, memory access restrictions over shared and private variables, as in section 3, can add to the complexity of the executing code.

Complexity in this instance comes from (at least) two key sources. Firstly, as with any evolved code, we run the risk of code growth, and, secondly, from the vagaries of scheduling what can be a very high number of threads. We first examine code growth.

6.1 Code Growth

We analyze the size of standard GE (without OpenMP pragmas; it evolves serial programs), and MCGE-II genotypes. GE genomes have two sizes [12]: actual and effective. Actual length is the total size of a genotype, while effective length is the part, used to generate the end program.

Figure 3 presents the actual and effective lengths of GE, MCGE-II (Unoptimized, Combined) averaged across 50 runs of 100 generations of *Fibonacci*. As expected, the actual and effective lengths vary significantly for the particular set-ups, with Wilcoxon Signed Rank Sum tests at $\alpha = 0.05$.

When we compare across approaches, surprisingly, the statistical tests show insignificant difference between the actual length of GE and MCGE-II (Unoptimized, Combined). In fact GE generates larger genotypes than required, thus, the MCGE-II grammars do not influence the actual length, as both use the same search engine. Instead, MCGE-II uses some of the actual size to map the OpenMP pragmas, hence, effective length increases. In GE, the actual lengths range from 25 to 47, while effective lengths range from 5 to 8 depending on the problem.

The **effective** lengths of MCGE-II (Unoptimized, Combined) are significantly larger than that of GE at $\alpha = 0.05$ due to the extra mapping steps. In MCGE-II, they range from 12 to 18, based on the problem. However, among the MCGE-II variants, the effective lengths do not vary significantly. It shows the fact that the changes in the design of MCGE-II grammars forfeit the overhead in code growth.

This slight increase in effective length may impact the performance of the programs only for 2 cores as is manifested in [7], hence, performance suffers (shown in section 5). However, for 4 cores and above, it has negligible implications, because now the impact of the extra cores is greater than that of the extra code. Also note, the code growth is independent of the number of cores under execution.

However, it is interesting to observe that GE does not bloat like GP, a phenomenon that was also noted in [3]. And, the reasons behind such observations remain unanswered, leaving a lot of scope for future work.

6.2 Computational Complexity

We also empirically analyze the computational complexity of MCGE-II (Combined) generated task parallel recursive programs. In this analysis, we consider the number of recursive calls of a program over the given input.

Although, space limitations do not permit reporting the details, the analysis shows that the problems *Sum-of-N*, *Factorial*, *Reverse* exhibit $\mathrm{O}(n)$ (linear) complexity whereas, *Binary-Sum* and *Quick sort* exhibit $\mathrm{O}(logn)$ and, $\mathrm{O}(nlogn)$ complexity respectively, while *Fibonacci* shows $\mathrm{O}(2^n)$ complexity. This demonstrates that the evolving programs are competitive with that of the conventional human written programs. It is to be noted that the use of parallel hardware can only reduce the computational load by dividing it among the existing processing elements, but not over the computational complexity. To that end in this paper, *Binary-Sum* exhibits the lowest complexity ($\mathrm{O}(log(n))$).

However, this analysis suggests that Fibonacci required an exponential number of recursive calls. In such cases the performance fails to scale-up due to *excessive parallelism*, because an exponential number of recursive calls create an exponential number of threads, where too many threads operate to perform too little individually. Clearly, this means that the degree of parallelism needs to be managed better. Hence, next we further optimize the performance.

$$\langle condition \rangle ::= \text{if}(\langle input\rangle\langle lop\rangle\langle const\rangle)\text{`\{'}\langle newline\rangle \langle line1\rangle;\langle newline\rangle\langle line2\rangle;\langle newline\rangle\text{`\}'}$$

is altered to appear as

$$\langle condition \rangle ::= \text{if}(\langle input\rangle\langle lop\rangle\langle const\rangle)\text{`\{'}\langle newline\rangle \langle line1\rangle;\langle newline\rangle\langle line2\rangle;\langle newline\rangle\text{`\}'} \langle newline\rangle \text{ else if } (\langle input\rangle \langle lop\rangle \langle const\rangle\langle const\rangle) \text{ `\{' } \langle newline\rangle \langle line1\rangle; \langle newline\rangle \langle line2\rangle; \langle newline\rangle\text{`\}'}$$

Figure 4: The enhanced MCGE-II grammars to generate a program that is both serial and parallel.

```
if (n <= 2) { temp = n; res += temp; }
else if(n <= 39)
{ temp = fib(n-1)+fib(n-2); res += temp; }
else {
 #pragma omp parallel sections\
 private (a) shared(n, temp, res)
 { #pragma omp section
  { a = fib(n-1);
    #pragma omp atomic res += temp+a; }
   #pragma omp section
  { a = fib(n-2);
    #pragma omp atomic res += temp+a; }
} } return res;
```

Figure 5: Evolved program that combines both parallel and serial execution to increase the speed-up.

7. FURTHER PERFORMANCE SCALING

Armed with the knowledge from the previous section we seek to constrain the systems so as to reduce the chances of excessive parallelism. To do this, we combine parallel and serial implementations of the evolved programs, which, further improves the performance. This reduces the overhead caused due to excessive parallelism as the top level recursive calls distribute load across a number of threads, whereas the lower level recursive calls carry out appropriately sized chunks of work instead of merely invoking more threads. We leave it up to evolution to detect the appropriate *level* at which recursion switches from parallel to serial.

To replicate these changes, we enhance the MCGE-II grammars (in Figure 1) to appear as shown in Figure 4, termed

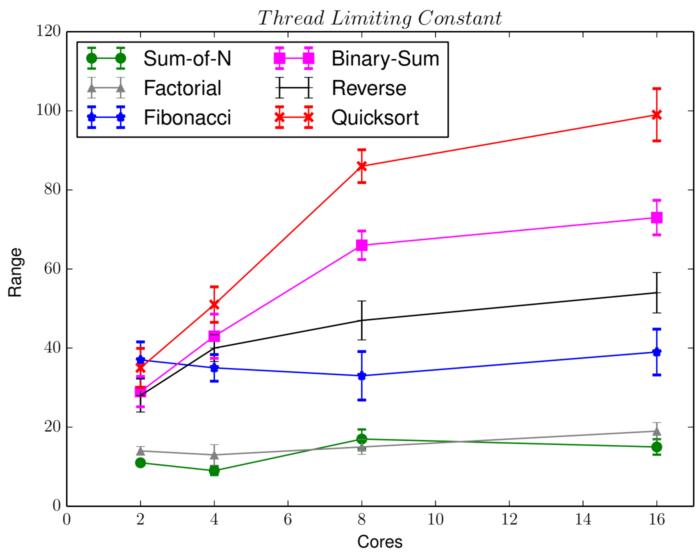

Figure 6: MCGE-II (Scaled) evolved thread limiting constants averaged across 50 runs.

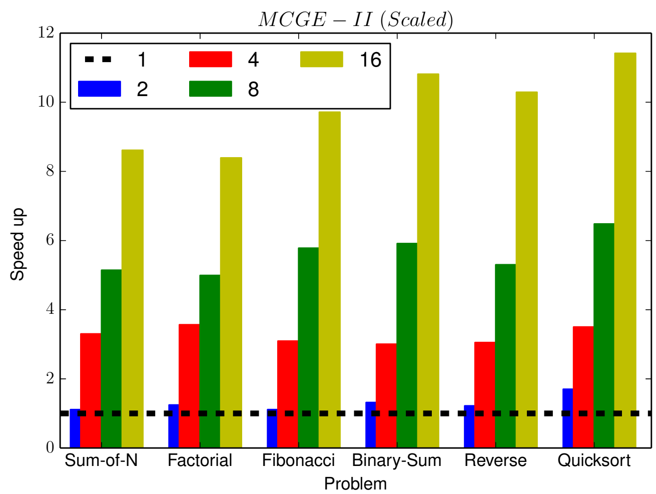

Figure 7: The performance of MCGE-II (Scaled).

Table 5: The mean best generation (mean ± [standard deviation]) of MCGE-II (Grammar, Combined, Scaled). The lowest value is in boldface.

#	MCGE-II Grammar	MCGE-II Combined	MCGE-II Scaled
	mean best generation	mean best generation	mean best generation
1	45.38 ± [2.81]	**43.27 ± [5.37]**	44.38 ± [2.81]
2	**31.19 ± [4.73]**	36.51 ± [3.67]	39.11 ± [4.73]
3	**44.73 ± [5.26]**	59.89 ±[4.15]	52.17 ± [4.45]
4	**59.14 ± [5.34]**	61.43 ± [5.19]	64.88 ± [3.51]
5	47.53 ± [2.19]	45.32 ± [4.92]	**44.53 ± [2.19]**
6	**40.49 ± [5.23]**	47.28 ± [3.15]	42.49 ± [5.23]

as *MCGE-II (Scaled)*, hereafter. We alter the non-terminal $<$ *condition* $>$ to generate nested *if-else* condition blocks. The changes generate a program that runs both in parallel as well as serial to reduce the execution time of the evolved programs. It also generates a two digit *thread limiting constant* automatically, at which, the serial execution is initiated.

Figure 5 shows the MCGE-II (Scaled) generated parallel recursive Fibonacci program. It adapts the thread limiting constant to the given problem and computational environment; this constant, arrests the further creation of threads and continues to execute serially. The intermediate result (*temp* in *else if*) is shared among the threads, thus, further optimizes the execution time, thereby, efficiently exploits the power of multi-cores.

The constant (39) in the *else if* (Figure 4) condition is the thread limiting constant for 16 cores. Figure 6 shows the thread limiting constants range (with standard deviation) with respect to the number of cores for all the six experimental problems. It evolves different limiting constants for each problem. Here, we evolve a two digit constant, which, for a large input (say, a 1000000 element array) may not be an optimal limitation, which might be a three or a four digit constant. This can be addressed with digit concatenation grammars [12, Chapter-5], a focus of the future research.

Figure 7 shows the speed-up of MCGE-II (Scaled) over all the six experimental problems. Overall, MCGE-II (Scaled) shows better performance over its counterparts.

Table 5 presents the *mean best generation* results of three MCGE-II variations. The results show that MCGE-II (Grammar) generates best programs faster than the two MCGE-II (Combined, Scaled) variants because of the grammatical bias, while the last two uses execution time in fitness evaluation, thus, makes the evolution slightly hard but generates the efficient task parallel recursive programs.

Table 6: Significance of *performance* and *mean best generation* of MCGE-II (Unoptimized, Grammar, Time, Combined, Scaled). Boldface shows the significance, while asterisk (*) shows the best variants.

MCGE-II variant	Average Rank	$p - value$	$p - Hommel$
Performance Optimization			
Unoptimized	**4.5**	**1.2604E-4**	**0.0125**
Grammar	**4.5**	**1.2604E-4**	**0.0166**
Time	**3.0**	**0.0284597**	**0.025**
Combined	**1.9998**	**0.0347332**	**0.05**
Scaled*	0.99999	-	-
Mean Best Generation			
Unoptimized	**4.333333**	**0.0019107**	**0.0125**
Grammar*	1.5	-	-
Time	**3.833333**	**0.0105871**	**0.01667**
Combined	**2.49998**	**0.0355132**	**0.05**
Scaled	**3.666667**	**0.0176221**	**0.025**

Table 6 shows the non-parametric Friedman tests with Hommel's post-hoc analysis on performance and mean best generation of MCGE-II (Unoptimized, Grammar, Time, Combined, Scaled). A variant with the lowest rank is the best among all of them, and is marked with an asterisk (*). Significantly different variants from the best variant at $\alpha = 0.05$

are in boldface, and is determined when the $p - value$ is less than p-Hommel. For *performance optimization*, MCGE-II (Scaled) outperforms the remaining four MCGE-II variants. Note, these results are for 16 cores of a processor, and are similar for the 8 cores, while they are insignificant with 4 cores and below. On average, for 16 cores, MCGE-II (Scaled) speeds up by a factor 9.97, which improves over MCGE-II (Combined) and MCGE-II (Unoptimized) by 17.45% and 37.91% respectively.

For *mean best generation*, the MCGE-II (Grammar) outperforms the remaining four MCGE-II variants. Note that these results are for 16 cores, while they are similar for 8 cores and below. Although MCGE-II (Scaled) requires slightly more number of generations over MCGE-II (Grammar, Combined), it is considered as the best variant as it generates efficient parallel programs.

However, a similar solution to avoid the inefficiency by the recursive calls is by keeping a table that records the result of a recursive call in its first evaluation. Then, we can refer the table for the repeated recursive calls, similar to Koza [9]. But this approach has often been criticized [10] in the EC community for not being an exact recursion.

8. CONCLUSION AND FUTURE WORK

In summary, we extended MCGE-II to automatically generate efficient task parallel recursive programs. This study offered a separation between the task and data parallelism in the design of the grammars along with the execution time in fitness evaluation. The modifications in the grammar favoured quick generation of programs, while the execution time helped in optimizing their performance.

We then analysed the effect of OpenMP thread scheduling and code growth on performance. Scheduling issues cause performance implications, while code growth (except for 2 cores) has negligible effect on performance of the evolving parallel programs. We also, analysed the computational complexity of the evolving programs, where excessive parallelism restricted the degree of parallelism in the evolving programs. We limited this behaviour with the evolution of programs that run both in serial (for lower level recursive calls) and parallel, thus, further optimized the performance.

Although the limiting constant that switches between parallel and serial modes of execution fails to generalize for large input because we limit it to two digits, it is easy to amend the grammar so that it can evolve a larger or smaller constant as the problem demands.

9. REFERENCES

[1] A. Agapitos and S. M. Lucas. Evolving efficient recursive sorting algorithms. In *IEEE Congress on Evolutionary Computation*, pages 2677–2684, 2006.

[2] A. Agapitos and S. M. Lucas. Evolving modular recursive sorting algorithms. In M. Ebner et.al., editor, *EuroGP 2007*, volume 4445 of *LNCS*, pages 301–310. Springer, Heidelberg, 2007.

[3] R. M. A. Azad and C. Ryan. The best things don't always come in small packages: Constant creation in grammatical evolution. In M. Nicolau et. al., editor, *EuroGP 2014*, volume 8599 of *LNCS*, pages 186–197. Springer, Heidelberg, 2014.

[4] S. Brave. Evolving recursive programs for tree search. In *Advances in Genetic Programming 2*, pages 203–220. MIT Press, 1996.

[5] G. Chennupati, R. M. A. Azad, and C. Ryan. Multi-core GE: automatic evolution of CPU based multi-core parallel programs. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1041–1044. ACM, 2014.

[6] G. Chennupati, R. M. A. Azad, and C. Ryan. Automatic evolution of parallel recursive programs. In *EuroGP 2015*, volume 9025 of *LNCS*, pages 167–178. Springer, Heidelberg, 2015.

[7] 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.

[8] S. García and F. Herrera. An extension on "statistical comparisons of classifiers over multiple data sets" for all pairwise comparisons. *Journal of Machine Learning Research*, 9:2677–2694, 2008.

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

[10] A. Moraglio, F. E. B. Otero, C. G. Johnson, S. Thompson, and A. A. Freitas. Evolving recursive programs using non-recursive scaffolding. In *IEEE CongressonEvolutionaryComputation*, pages 1–8, 2012.

[11] A. Nisbet. GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In P. Sloot et. al., editor, *High-Performance Computing and Networking*, volume 1401 of *LNCS*, pages 987–989. Springer, 1998.

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

[13] OpenMP Architecture Review Board. OpenMP application program interface version 3.0. http://www.openmp.org/mp-documents/spec30.pdf, May 2008.

[14] C. Ryan. *Automatic Re-engineering of Software Using Genetic Programming*, volume 2 of *Genetic Programming*. Springer, 1999.

[15] L. Spector, J. Klein, and M. Keijzer. The push3 execution stack and the evolution of control. In *Proceedings of Genetic and Evolutionary Computation Conference*, pages 1689–1696. ACM, 2005.

[16] A. Trenaman. Concurrent genetic programming, tartarus and dancing agents. In R. Poli et. al., editor, *EuroGP 1999*, volume 1598 of *LNCS*, pages 270–282. Springer, Heidelberg, 1999.

[17] P. A. Whigham and R. I. McKay. Genetic approaches to learning recursive relations. In X. Yao, editor, *Progess in Evolutionary Computation*, LNCS, pages 17–27. Springer, Heidelberg, 1995.

[18] K. P. Williams. *Evolutionary algorithms for automatic parallelization*. PhD thesis, University of Reading, 1998.

[19] M. L. Wong and T. Mun. Evolving recursive programs by using adaptive grammar based genetic programming. *GPEM*, 6:421–455, 2005.