Synthesis of Parallel Iterative Sorts with Multi-Core Grammatical Evolution

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

Writing parallel programs is a challenging but unavoidable proposition to take true advantage of multi-core processors.

In this paper, we extend *Multi-core Grammatical Evolution for Parallel Sorting* (MCGE-PS) to evolve parallel iterative sorting algorithms while also optimizing their degree of parallelism. We use evolution to optimize the performance of these parallel programs in terms of their execution time, and our results demonstrate a significant optimization of 11.03 in performance when compared with various MCGE-PS variations as well as the GNU GCC compiler optimizations that reduce the execution time through code minimization.

We then analyse the evolutionary (code growth) and non-evolutionary (thread scheduling) factors that cause performance implications. We address them to further optimize the performance and report it as 12.52.

Categories and Subject Descriptors

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

Keywords

Grammatical Evolution; Multi-cores; Automatic Parallelization; Performance Optimization; OpenMP; Sorting.

1. INTRODUCTION

As the number of cores on a single chip increases, programming those processors becomes increasingly difficult. For example, Intel Polaris and picoChip have 80 and 200+ cores, respectively. As a result, with the so-called *death of scaling*[1], they need to be programmed explicitly, to fully optimize the performance. Such optimization often requires the knowledge of hardware environment.

An elegant fix for this problem is to automatically generate computer programs with as little human intervention as possible. Recently, MCGE-PS [6] generated *natively* parallel sorting, using both Grammatical Evolution (GE) [19] and OpenMP [15] that also solved the underlying problem. Although these programs have reported better performance than their sequential counterparts, little attempt was made in analyzing their efficiency, particularly in terms of the degree of parallelism and execution time of these programs.

In this paper, we extend MCGE-PS to enhance the performance of the evolving parallel sorting programs, automatically. Similar to MCGE-PS, here, we use OpenMP with problem specific GE grammars. However, the grammars are designed (as opposed to [6]) so as to offer greater flexibility in evolving a parallel program. Then, this work optimizes the performance primarily through aptly selecting an OpenMP *pragma* (a pre-processor directive for parallelization) by considering execution time of an evolved program in its fitness evaluation. We assess this on four benchmark sorting problems in C as they are suitable for parallelization.

We compare the performance among various MCGE-PS variations and three GNU GCC compiler optimization flags that try to reduce the execution time of a program automatically. The results demonstrate a significant speed-up of 11.03 (in terms of execution time) when the evolving parallel programs are executed on 16 cores of an Intel processor.

We then analyze the effect of *code growth* and *OpenMP thread scheduling* on performance. We observe that the code growth has negligible effect except for 2 cores, while scheduling poses performance challenges with its load balancing. That is, the ideal *chunk size* (the amount of work divided among the threads) varies with the number of cores, amount of work, and the number of threads under execution. We resolve this issue by automatically evolving the *chunk size*. As a result, we further optimize the performance to 12.52.

The rest of the paper is laid out as follows: section 2 discusses the literature in the context of this paper; section 3 explains the proposed approach; section 4 describes the experiments; section 5 shows the results; section 6 discusses the performance bottlenecks; and finally, section 7 concludes.

2. BACKGROUND

We describe the evolution of sorting in section 2.1, and in section 2.2, we review automatic parallel code generation.

2.1 Evolutionary Techniques for Sorting

In evolving sorting, Hillis [8] efficiently evolved a minimal 16-input network for the sorting network problem. O'Reilly and Oppacher [16] initially failed to evolve sorting with genetic programming (GP); however, they succeeded in [17]

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

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

DOI: http://dx.doi.org/10.1145/2739482.2768458

with a *swap* primitive. Later, Kinnear [9, 10] generated a bubble sort by swapping the disordered adjacent elements.

Abbott [1] used Object Oriented Genetic Programming (OOGP) for insertion and bubble sorts. Spector et al., [21] used PushGP for recursive sorting that had an $O(n^2)$ complexity and enhanced to $O(nlog(n))$ by adding efficiency.

Recently, Agapitos and Lucas [2, 3] evolved efficient recursive quick sort using OOGP in Java. The evolved sorting programs were of $O(nlogn)$ complexity. Then, O'Neill et al. [13] applied GE for program synthesis by evolving an iterative (using *for* loops) bubble sort in Python; the evolved programs had quadratic $O(n^2)$ complexity.

Most of these attempts belong to quadratic complexity $O(n^2)$, while the attempts in [2, 21] belongs to $O(nlogn)$.

2.2 Automatic Parallel Code Generation

In general, automatic parallel code generation can be classified as either *auto-parallelization of serial code* or *native parallel code generation*. The former parallelizes an *existing* sequential program that works correctly, while the latter generates a working program which is *also* parallel.

In *auto-parallelization* with GP, first Walsh and Ryan introduced *Paragen-I* [18, Chapter-5] to map serial programs onto multiprocessors. Then, *Paragen-II* [18, Chapter-7] dealt transformations in *Atom* and *Loop* modes. *Atom* dealt simple instructions while *Loop* dealt loop sequences. Later, Ryan and Ivan [20] extended *Paragen-II* to merge independent tasks of different loops into a single loop.

With genetic algorithms (GA), Nisbet [12] introduced Genetic Algorithm Parallelization System (*GAPS*) for sequence restructuring. Then, Williams in *Revolver* [25] optimized the execution time with program and loop transformations.

For the attempts in *native parallel code generation*, Trenaman [22] concurrently executed autonomous agents in the design of controllers for virtual world using multi-tree GP.

Recently, [5] automatically evolved the parallel regression programs on multi-cores that accelerated the program generation. Then, [6] evolved parallel sorting algorithms. However, efficiency of such programs is often questionable. To that end, in this paper, we optimize their performance.

3. MCGE-PS

We enhance MCGE-PS, that offers greater flexibility with the design changes in the grammars (as opposed to [6]). That include OpenMP private, shared and scheduling clauses in the evolving parallel programs. Then, the fitness function reduces their execution time assuring the efficiency. However, it still uses *single program multiple data (SPMD)* parallelization.

3.1 Design of Grammars

The selection of an appropriate pragma is crucial to the overall performance of the programs, while it is equally important for their quick generation. We achieve this automatically by separating the data and task parallel pragmas.

Figure 1 shows the MCGE-PS grammars to evolve a parallel *Odd-Even sort*, that works by swapping the adjacent elements in two phases. The non-terminal `<omppragma>` has separate rules for task (`<omptask>`) and data parallelism (`<ompdata>`); evolution selects one of them. The best evolved programs prefer the `<ompdata>` pragmas.

Here, the input (`<var>`), index (`<index>`), and the size (*length*) of the array are shared among all the cores. We use

```
⟨program⟩       ::= ⟨for_out⟩ ⟨newline⟩ ⟨condition⟩

⟨condition⟩     ::= if(⟨index⟩ ⟨bop⟩ ⟨const⟩ ⟨lop⟩ ⟨const⟩)
                    '{' ⟨ompprogram⟩ '}' ⟨newline⟩ else '{'
                    ⟨ompprogram⟩'}'

⟨ompprogram⟩    ::= ⟨newline⟩ ⟨omppragma⟩
                    ⟨sharedprivate⟩ ⟨schedule⟩ ⟨newline⟩
                    ⟨for_in⟩ ⟨newline⟩

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

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

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

⟨sharedprivate⟩ ::= shared(⟨var⟩, ⟨index⟩, length) ⟨private⟩
                    ⟨newline⟩ '{' ⟨newline⟩

⟨private⟩       ::= private(⟨index⟩)
                  | firstprivate(⟨index⟩)
                  | lastprivate(⟨index⟩)

⟨schedule⟩      ::= schedule(⟨type⟩, CHUNK)

⟨type⟩          ::= static | dynamic | guided

⟨for_out⟩       ::= for(i=0; i < length; i++) '{'

⟨for_in⟩        ::= for(j=1; j < length-1; j+=2) '{'
                    ⟨newline⟩ ⟨for_in_line⟩ ⟨newline⟩ '}'

⟨for_in_line⟩   ::= if(⟨var⟩[abs(⟨index⟩ ⟨bop⟩ ⟨const⟩)]
                    ⟨lop⟩ ⟨var⟩[abs(⟨index⟩ ⟨bop⟩ ⟨const⟩)])
                    '{'⟨newline⟩ ⟨swap⟩ '}'

⟨swap⟩          ::= temp = ⟨var⟩[abs(⟨index⟩
                    ⟨bop⟩ ⟨const⟩)]; ⟨newline⟩
                    ⟨var⟩[abs(⟨index⟩ ⟨bop⟩ ⟨const⟩)]
                    = ⟨var⟩[abs(⟨index⟩ ⟨bop⟩ ⟨const⟩)];
                    ⟨newline⟩ ⟨var⟩[abs(⟨index⟩ ⟨bop⟩
                    ⟨const⟩)]=temp; ⟨newline⟩

⟨bop⟩           ::= + | -

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

⟨const⟩         ::= 0 | 1

⟨index⟩         ::= i | j | temp

⟨var⟩           ::= A

⟨newline⟩       ::= \n
```

Figure 1: MCGE-PS grammars that evolve a natively parallel iterative *Odd-Even sort* algorithm.

the adjacent element *swap* (`<swap>`) in solving the problem. The temporary variable (*temp* in `<index>`) is private to the thread under execution. The production rules of `<private>` represents the three OpenMP private clauses. Of which, *private* allows variable read/write operations private to the thread, *firstprivate* keeps initial value of a variable irrespective of the parallel region (used to explicitly port an external value to the parallel region), while *lastprivate* holds the last change of a variable in the parallel region. However, the last two rules of `<private>` generate a bad individual as *temp* in `<swap>` holds a different element in each iteration. Hence, evolution keeps *private (<index>)* clause in the best evolved program through its fitness evaluation.

Similarly, in scheduling (`<type>`) the parallel (`<for_in>`) loop, OpenMP offers three clauses: *static*, *dynamic* and *guided*. Of which, *static* divides the work among threads before the loop execution; *dynamic* allocates the work during the execution; *guided* also divides the work during the execution but, the allocation begins with the large *chunk size* and decreases for the next requests. These clauses operate on a default *chunk size* of 1, we use *chunk=10*. A study on an ideal *chunk* is laid out later in section 6.1. As the time varies with the schedule type, a program with *dynamic* clause is the best fit than the ones with other schedule clauses.

The grammar also allows binary operations (`<bop>`, `<lop>`) and constants (`<const>`). Since C/C++ prohibits negative indexing of an array, we use the absolute values (*abs*) as shown in `<for_in_line>` and `<swap>`. An example of a best evolved parallel iterative Odd-Even sort is in section 6.3.

3.2 Performance Optimization

Since use of different pragmas can alter performance, the execution time of programs vary. Thus, we take into account the execution time to compute the fitness of a program.

Thus, the fitness function is a product of the execution time and the program accuracy. The program accuracy is measured in terms of mean *inversions* (pairs that are out of order). For example, if $a_1 a_2 a_3 \ldots a_n$ is a permutation of the set $1, 2, \ldots, n$ then the pair (a_i, a_j) is an *inversion*[11] of the permutation iff $i < j$ and $a_i > a_j$. Both the fitness components are normalized in the range $(0, 1)$ – maximization function. Then, the fitness function (f_{pprog}) is as follows:

$$f_{pprog} = \frac{1}{(1+t)} * \frac{1}{\left(1 + \frac{\sum_{i=1}^{N} n\big(I(A_i)\big)}{T.P}\right)} \tag{1}$$

where, t stands for the total execution time of the evolved parallel program over all the training cases (N); $n\big(I(A_i)\big)$, is the number of inversions in the i^{th} array (A_i; total, N arrays); and $T.P$ is the total number of pairs in all the training cases (N). Note that a training case is an array of elements.

Note, selecting a less than ideal pragma raises the execution time of the evolved parallel program. The time component of f_{pprog}, thus ensures to select an apt pragma. Meanwhile, *normalized mean inversions* assures the accuracy of sorting. Thus, the collective aim is to obtain a correct sorting program that is optimized for the multi-core processor.

4. EXPERIMENTS

We evaluate our approach on four iterative sorting algorithms; Table 1 presents these problems, detailing the type of input (*int*), number of arguments and the number of local variables (LV) for each problem. Their solutions use conditional (*if*), iterative (*for*) and variable indexing structures. We use 100 training cases with a 1000 elements array at each case, that are randomly generated from the range [1 : 1000].

Table 1: The problems and the local variables (LV).

#	Problem	Input	LV	Range
1	Bubble sort	int [ ], int	4	[1:1000]
2	Quick sort	int [ ], int, int	5	[1:1000]
3	Odd-Even sort	int[ ], int	4	[1:1000]
4	Rank sort	int [ ], int	4	[1:1000]

Table 2: Parameters, experimental environment.

Algorithmic parameter settings	
Parameter	**Value**
point mutation	0.01
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*()

Table 2 describes the GE parameters along with the hardware and software specifications on Intel processor.

We divide the experiments into two sets. The first set investigates the performance of different MCGE-PS variants. The aim is to analyze the effect of the design of grammars, and the fitness evaluation (eq. 1) on performance. The second set compares the performance of the evolved parallel programs with the compiler optimizations in terms of execution time. Therefore, this study shows the performance of MCGE-PS evolving parallel programs. The experimental settings in Table 2 are consistent for all the experiments.

4.1 MCGE-PS Variations

We report the speed-up of four MCGE-PS variants that vary in the design of the grammars, and fitness evaluation. The first variant named *MCGE-PS (Unoptimized)* hereafter, does not offer any separation between the task and data parallel pragmas; rather, the rules in `<omptask>` and `<ompdata>` work together under `<omppragma>`. Thus it is hard for the evolution to omit task parallelism when it only requires data parallelism, and normalized mean inversions for fitness evaluation. The second variant, *MCGE-PS (Grammar)*, uses the design of the grammars shown in section 3.1, fitness evaluation is the normalized mean inversions. The

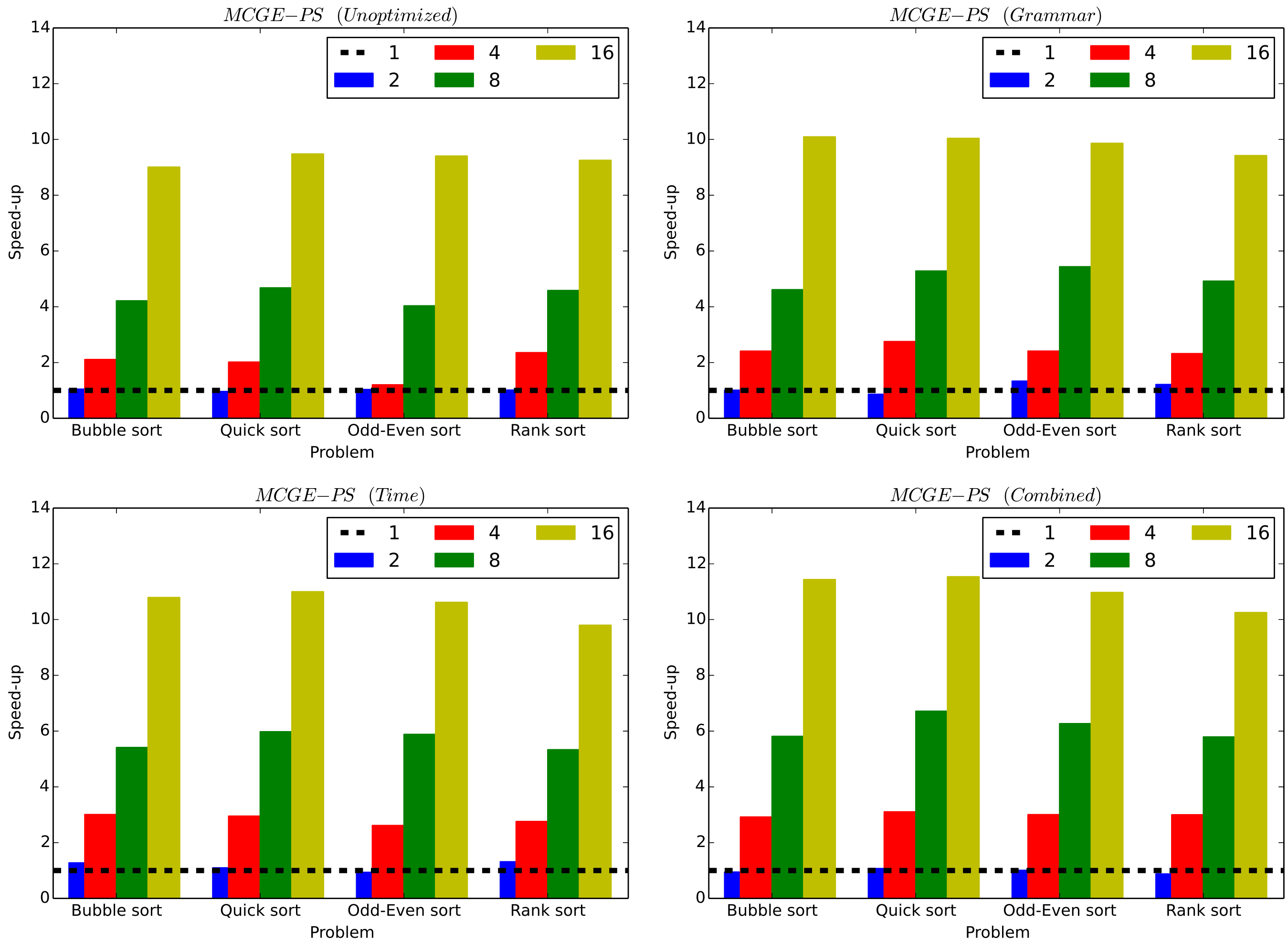

Figure 2: The performance (speed-up) of all the four MCGE-PS (Unoptimized, Grammar, Time, Combined) variants evolved parallel programs over all the four experimental problems with varying number of cores (2, 4, 8, and 16). The horizontal dashed (− −) line represents the speed-up of 1 (ideally, speed-up of uni core).

Table 3: Friedman statistical tests with Hommel's post-hoc analysis on the performance (speed-up) of four MCGE-PS variants when the number of cores is 4, 8 and 16. The boldface represents the significantly different results at $\alpha = 0.05$, while asterisk (*) indicates the best variant.

MCGE-PS variant	Average Rank	p value	p Hommel
4 cores			
Unoptimized	**3.75**	**0.006169**	**0.0166**
Grammar	3.25	0.028459	0.025
Time	1.75	0.58388	0.05
Combined*	1.25	-	-
8 and 16 cores			
Unoptimized	**4.0**	**0.001015**	**0.0166**
Grammar	**3.0**	**0.0204597**	**0.025**
Time	2.0	0.2733216	0.05
Combined*	1.0	-	-

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

#	MCGE-PS Unoptimized mean best generation	MCGE-PS Grammar mean best generation	MCGE-PS Time mean best generation	MCGE-PS Optimized mean best generation
1	67.19 ± [4.16]	**37.63** ± **[3.19]**	73.27 ± [3.31]	41.27 ± [0.81]
2	47.61 ± [3.51]	**31.35** ± **[3.65]**	51.51 ± [3.67]	33.49 ± [2.95]
3	58.69 ± [5.86]	44.19 ± [6.43]	62.89 ±[4.15]	**35.27** ± **[3.46]**
4	54.11 ± [3.43]	**29.88** ± **[4.51]**	61.43 ± [5.19]	31.14 ± [3.17]

Friedman statistical tests with Hommel's post-hoc analysis. Boldface represents the significance at $\alpha = 0.05$, while asterisk (*) shows the best variant among all the four MCGE-PS variants.

MCGE-PS variant	Average Rank	p-value	p-Hommel
Unoptimized	**3.25**	**0.0284597**	**0.025**
Grammar*	1.25	-	-
Time	**3.75**	**0.0061698**	**0.0166**
Combined	1.75	0.5838824	0.05

Table 5: The *mean best execution time* (in secs) (mean [standard deviation]) of MCGE-PS (Unoptimized, Combined) and the optimization flags (O1, O2, O3). The boldface represents the lowest execution time.

Cores	Problem	Performance				
		O1	O2	O3	MCGE-PS (Unoptimized)	MCGE-PS (Combined)
2	Bubble sort	3917.96 [29.91]	3592.07 [25.44]	**3166.32 [23.07]**	3334.08 [31.51]	3687.01 [9.17]
	Quick sort	4677.91 [29.13]	4713.52 [28.49]	4623.43 [29.31]	4543.43 [18.12]	**4476.79 [20.21]**
	Odd-Even sort	3498.46 [36.76]	3339.13 [30.19]	3431.29 [28.34]	**3096.46 [22.11]**	3175.78 [32.43]
	Rank sort	3188.56 [23.42]	2397.19 [29.39]	**2098.85 [29.17]**	3158.35 [21.33]	3144.69 [26.92]
4	Bubble sort	4101.39 [21.46]	3092.11 [29.44]	2396.32 [29.25]	1285.88 [14.24]	**1097.75 [16.56]**
	Quick sort	3987.39 [27.23]	3999.17 [29.36]	3819.36 [21.92]	**1448.06 [19.17]**	1535.53 [22.14]
	Odd-Even sort	3219.54 [24.24]	3331.57 [31.26]	2883.71 [22.19]	2664.21 [24.28]	**1032.94 [29.53]**
	Rank sort	2744.72 [29.59]	2234.62 [30.17]	2584.53 [29.33]	**1080.97 [27.35]**	1164.92 [19.44]
8	Bubble sort	2931.09 [19.59]	2763.73 [26.35]	2636.44 [28.79]	668.23 [13.18]	**551.34 [15.72]**
	Quick sort	2795.54 [27.36]	2293.24 [25.78]	2829.57 [25.36]	768.99 [20.37]	**648.08 [21.34]**
	Odd-Even sort	2262.54 [31.42]	3109.34 [28.27]	2396.23 [29.17]	635.23 [26.29]	**511.65 [29.19]**
	Rank sort	2435.35 [22.34]	2319.94 [33.37]	2445.52 [29.23]	**571.91 [31.11]**	543.35 [27.52]
16	Bubble sort	2888.49 [11.86]	2996.78 [19.01]	2541.54 [16.45]	386.588 [19.33]	**307.18 [16.93]**
	Quick sort	2226.29 [24.65]	2173.26 [12.31]	2123.65 [16.52]	460.59 [15.21]	**367.62 [19.48]**
	Odd-Even sort	2347.13 [25.40]	2835.84 [27.32]	3177.11 [24.81]	339.92 [23.25]	**292.27 [27.11]**
	Rank sort	2703.21 [35.42]	2698.76 [19.14]	2285.86 [12.61]	345.12 [29.42]	**310.66 [22.62]**

third variant, *MCGE-PS (Time)*, uses the design of grammars in [6], evaluates the fitness with f_{pprog} (eq. 1). Then, the fourth variant, *MCGE-PS (Combined)*, combines the design of the grammars shown in section 3.1 and the fitness function f_{pprog}. Therefore, the objective is to show how MCGE-PS achieves the twin objective of program correctness and performance optimization.

5. RESULTS

We report the performance of MCGE-PS in terms of *mean best execution time*: the total execution time of all the best of generation programs of a run; averaged across 50 runs. In section 4.1, we compare the speed-up (the ratio of mean best execution time on n-cores to 1-core) of different MCGE-PS variants. Then, section 5.1 compares the mean best execution time of MCGE-PS with that of compiler optimizations.

Figure 2 shows the speed-up of MCGE-PS (Unoptimized, Grammar, Time, Combined) on all the four problems. On an average over all the problems, MCGE-PS (Combined) shows a speed-up of 11.03, a better improvement of 15.75% over MCGE-PS (Unoptimized) that has a speed-up of 9.29.

Table 3 shows the non-parametric Friedman statistical tests with Hommel's post-hoc analysis [7] on performance of MCGE-PS (Unoptimized, Grammar, Time, Combined). The first column shows the MCGE-PS variant; second column shows the average rank; third column presents the *p-value*; the fourth column shows the *p-Hommel* of the *post-hoc* analysis. A variant with the lowest average rank is the best variant (MCGE-PS(Combined)) and marked with an asterisk (*). A value is in *boldface* if it is significantly different from the best variant; That is the *p-value* of the corresponding method is less the critical *p-Hommel* at $\alpha = 0.05$.

The results are insignificant for 2 cores[2] among all the four MCGE-PS variants (reasons are in section 6). For 4 cores, MCGE-PS (Combined) outperforms MCGE-PS (Unoptimized) while it is insignificant from MCGE-PS (Grammar, Time). For 8 and 16 cores, MCGE-PS (Combined) outperforms MCGE-PS (Unoptimized, Grammar), and is insignificant over MCGE-PS (Time). The reasons are better justified with the program generating ability of MCGE-PS.

We now compare the *mean best generation* of MCGE-PS (Unoptimized, Grammar, Time, Combined). *Mean best generation* is the number of generations taken by MCGE-PS in evolving the best parallel program, that is averaged across 50 runs. Table 4 shows the mean best generation of MCGE-PS (Unoptimized, Grammar, Time, Combined) and their statistical significance results. The results indicate that MCGE-PS (Grammar) outperforms MCGE-PS (Unoptimized, Time) while it is insignificant over MCGE-PS (Combined). In other words, MCGE-PS (Grammar) (the changes in the design of grammars alone) produces parallel iterative sorting programs in fewer generations, while MCGE-PS (Time) takes more generations. It is because of the alterations in the design of the grammars among MCGE-PS variants, a phenomenon similar to [24], effects the evolution of the programs. However, MCGE-PS (Combined) evolves efficient parallel sorting with an insignificant difference with MCGE-PS (Grammar). Hence, MCGE-PS (Combined) is the best variant for the automatic evolution of efficient parallel sorting programs.

5.1 Compiler Optimizations

Compiler optimizations[3] (–O1, –O2, –O3) try to minimize the code, and reduce the execution time. Of these flags, $O1$, optimizes the source code with conditional branching, copy propagations, etc and moderately reduces the time with no changes in compile time. $O2$, along with $O1$, offers aliasing, cross jumps, etc to fully reduce the time with a slight increase in compile time. $O3$, along with $O2$, offers autovectorization, function in-lining, etc to fully reduce the execution time. To that end, GE evolves serial programs[4] with

[2]Note that the significance results for 2 cores are not provided in Table 3 due to space constraints.

[3]https://gcc.gnu.org/onlinedocs/gcc-4.4.5/gcc/Optimize-Options.html

[4]Neglecting parallelism exerting non-terminals from Figure 1 evolves the serial sorting programs.

these flags. We then compare their execution time with that of MCGE-PS (Unoptimized, Combined).

Table 5 compares the performance among the two MCGE-PS variants and the three optimization flags for 2, 4, 8, and 16 cores. The lowest execution time is in boldface for the corresponding method on a given problem. Although the programs evolved with optimization flags reduce the execution time, MCGE-PS variants exhibit better optimization.

Table 6: Significance of *performance* of MCGE-PS (Unoptimized, Combined) and optimization flags (O1, O2, O3) at $\alpha = 0.05$. The best method is highlighted with asterisk (*), while the methods that are significantly different from the best are in boldface.

Cores	Method	Average Rank	p value	p Hommel
4	O1	**4.5**	**0.00729**	**0.0125**
	O2	**4.5**	**0.00729**	**0.0166**
	O3	**2.75**	**0.02357**	**0.025**
	Unoptimized	**2.15**	**0.02306**	**0.05**
	Combined*	1.5	-	-
8	O1	**5.0**	**3.65E-3**	**0.0025**
	O2	**4.0**	**0.00961**	**0.0196**
	O3	**2.5**	**0.0107**	**0.020**
	Unoptimized	**2.25**	**0.01952**	**0.025**
	Combined*	1.75	-	-
16	O1	**4.85**	**7.32E-4**	**0.0012**
	O2	**4.15**	**0.00254**	**0.0107**
	O3	**2.5**	**0.00134**	**0.020**
	Unoptimized	**1.95**	**0.02012**	**0.025**
	Combined*	1.15	-	-

Table 6 shows the non-parametric Friedman statistical tests with Hommel's post-hoc analysis on performance. The best approach (MCGE-PS (Combined)) is marked with an asterisk (*). A value is in *boldface* if it is significantly different from the best method. The results indicate that MCGE-PS (Combined) outperforms all its counterparts except for 1, and 2 cores. Although it is expected that increasing the number of cores should reduce the time to execute a parallel program, that is not true for all these results. Instead, MCGE-PS (Combined) outperforms MCGE-PS (Unoptimized) when the number of cores is greater than 2.

Finally, MCGE-PS (Combined) evolving parallel sorting programs record better performance over the GE evolving optimized serial sorting programs. Next, we analyze the factors that influence the performance.

6. DISCUSSION

In this section, we discuss two major factors that impact the performance of the evolved programs, that is, OpenMP scheduling (section 6.1) and code growth (section 6.2).

6.1 Performance Bottlenecks

The non-evolutionary factors such as OpenMP scheduling play a vital role in optimizing the performance. Interestingly, OpenMP hides these details from the developer, which makes it easy to use, at the same time hard to realize its full potential. Load balancing by parallel threads is a serious concern on shared memory processors. OpenMP scheduling strategies (*static*, *dynamic*, *guided*) (described earlier in section 3.1) answer these performance issues effectively. However, it becomes complicated in setting the optional *chunk size* (*chunk*) explicitly, as the ideal value often requires the problem specific knowledge. That is, it changes with respect to the amount of work (loop iterations), number of cores and the threads under execution.

We overcome this by evolving an appropriate *chunk size* irrespective of the problem and the number of cores that it executes. We adopt the digit concatenation grammars that are used in solving the symbolic regression problems.

$$\langle \mathit{schedule} \rangle \quad ::= \text{schedule}(\langle \mathit{type} \rangle, \text{CHUNK})$$

is modified to appear as

$$\langle \mathit{schedule} \rangle \quad ::= \text{schedule}(\langle \mathit{type} \rangle, \langle \mathit{const1} \rangle)$$

$$\begin{aligned} \langle \mathit{const1} \rangle \quad ::= &\ 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \mid &\ \langle \mathit{const1} \rangle \langle \mathit{const1} \rangle \end{aligned}$$

Figure 3: Enhanced MCGE-PS grammars that generate an adaptable *chunk size* for thread scheduling.

Figure 3 shows the modified MCGE-PS grammar that automatically generates a sequence of digits. The evolved *chunk size* adapts to the number of cores, amount of load, and the number of threads under execution. As a result, the evolved constant for *chunk size* balances the load effectively, thus, improves the performance. We report the speed-up of the enhancements, termed as MCGE-PS (Chunk), hereafter.

Figure 4 shows the *speed-up* of MCGE-PS (Chunk) evolved programs. It shows an average speed-up of 12.52 for 16 cores, a better improvement of 11.91% over MCGE-PS (Combined), an improvement of 25.79% over MCGE-PS (Unoptimized).

Table 7 represents the Wilcoxon statistical significance tests between MCGE-PS (Chunk) and MCGE-PS (Combined) at $\alpha = 0.05$. It contains the p-value for the corresponding problem while *"Yes"* states that the difference between the results of both the methods is significant; i.e., $p < 0.05$. Vargha-Delaney [23] A measure states how often that MCGE-PS (Chunk) outperforms MCGE-PS (Combined). A measure lies in between 0 and 1: when it is above 0.5, MCGE-PS (Chunk) is better than MCGE-PS (Com-

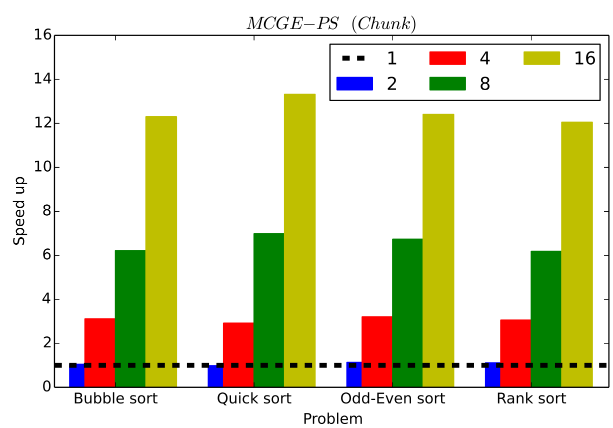

Figure 4: Performance of *MCGE-PS (Chunk)*

Table 7: Significance tests (at $\alpha = 0.05$) show that MCGE-PS (Chunk) outperforms MCGE-PS (Combined) for 8 and 16 cores. Note that "Yes" states the results are significant (p-value < 0.05). A measure shows the probability at which, MCGE-PS (Chunk) is better over MCGE-PS (Combined).

Cores	#	Wilcoxon Signed Rank Sum Test			A measure
		Rank Sum	p value	Significant	
8	1	2089	0.00632	Yes	0.6183
	2	2798	0.03183	Yes	0.3917
	3	3321	0.01119	Yes	0.7392
	4	2479	0.04178	Yes	0.2851
16	1	2250	0.04261	Yes	0.3545
	2	2701	0.00018	Yes	0.8751
	3	3253	0.00461	Yes	0.6559
	4	2221	0.03516	Yes	0.5215

bined); when it is 0.5, then both are equal; when it is less than 0.5 MCGE-PS (Combined) is better than MCGE-PS (Chunk); if it is close to 0.5 then the difference between them is small, otherwise the difference is large. For example, on *Bubble sort* with 16 cores, 35% of the time, MCGE-PS (Chunk) performs better than MCGE-PS (Combined). In other words, 65% of the time, MCGE-PS (Combined) performs better than MCGE-PS (Chunk). Overall, MCGE-PS (Chunk) performs better than MCGE-PS (Combined).

Table 8: MCGE-PS (Chunk) evolved *chunk size* (mean $\pm$ [standard deviation]), averaged across 50 runs for all the four experimental problems.

Cores	Problem	*chunk size* (CHUNK)
8	Bubble sort	135.17 ± [18.39]
	Quick sort	159.34 ± [22.71]
	Odd-Even sort	166.81 ± [17.33]
	Rank sort	142.53 ± [21.45]
16	Bubble sort	55.43 ± [10.62]
	Quick sort	67.91 ± [13.37]
	Odd-Even sort	80.15 ± [12.59]
	Rank sort	74.58 ± [11.11]

Table 8 shows the MCGE-PS (Chunk) evolved *chunk size*. It is the average of the evolved best of run programs averaged across 50 runs. The *chunk* results are reported for 8 and 16 cores. They showed significant performance optimization while, 2 and 4 are insignificant, hence, neglected.

Although the reduction in the execution time is significant, it does not reach ideal, owing to the Linux kernel scalability issues.

6.2 Code Growth

Given the importance of efficient code in parallel programs, the sort of code growth often associated with GP becomes more important, as it can impact the end product (parallel program), and the process to produce the code.

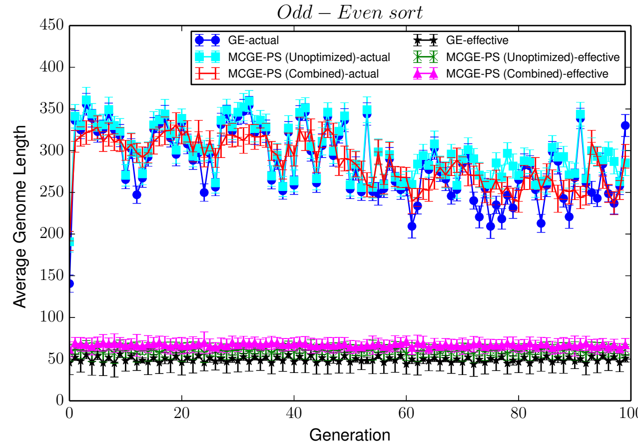

Figure 5: The actual and effective lengths of GE, MCGE-PS (Unoptimized, Combined) for *Odd-Even sort*. They are similar for the remaining problems and are not shown due to space constraints.

We compare the size of the evolving individuals of MCGE-PS (Unoptimized, Combined) and GE. A GE individual has two different lengths [14]: *actual*, *effective*. The actual length is the total size of the genotype, while effective length is the part of the total size used in mapping into a program.

Figure 5 shows the lengths of GE, MCGE-PS. As is typical to GE [14], both the lengths differ significantly (Wilcoxon Signed Rank tests at $\alpha = 0.05$) in a given approach.

Surprisingly, there is no significant difference between the actual lengths of GE and MCGE-PS. We hypothesis this as, GE generates larger genotypes than the required, that are unaffected even with the parallelization pragmas. Rather, a part of the genotype generates a parallel program, as a result, the effective length increases. The effective lengths of GE and MCGE-PS differ significantly at $\alpha = 0.05$ due to the fact that MCGE-PS requires extra number of mappings to evolve a parallel program. However, the effective size differs insignificantly on both the MCGE-PS variants at $\alpha = 0.05$.

Although, the effective lengths increase significantly, the difference is only marginal; it may offset the gains for 1 to 2 cores. However, for 4 cores and above, this increase is not much of an issue given the power of the processing elements.

However, we find that GE does not bloat as much as GP, a happening in [4] also. The reasons for such nature requires further analysis, a matter of future research.

6.3 Evolved Program

This section presents the evolved best parallel iterative sorting program, and its time complexity. Figure 6 presents the MCGE-PS (Chunk) evolved parallel iterative Odd-Even sort algorithm. Note, the program has two different *chunk* values (89, 87) as it operates in two phases (odd and even).

The empirical analysis on time complexity is performed in terms of the amount of time taken by the best evolved program for sorting an input. Paralleling an algorithm does not alter the complexity, nevertheless, it optimizes the time.

The results are abstracted out due to space restrictions. However, the complexity of these programs is competitive with the evolutionary attempts. Overall, *Quick sort* has the

```
for(i=0; i < length; i++) { if (i%2 == 0) {
 #pragma omp parallel for shared(A,length)\
  private(j,temp) schedule(dynamic, 89)
  for(j=1; j < length-1; j+=2) {
   if(A[abs(j-1)]<A[abs(j-0)]) {
    temp=A[abs(j-1)];A[abs(j-1)]=A[abs(j-0)];
    A[abs(j-0)]=temp; } } } else {
 #pragma omp parallel for shared(A,length)\
  private(j,temp) schedule(dynamic, 87)
  for(j=1; j < length-1; j+=2) {
   if(A[abs(j)] > A[abs(j+1)]) {
    temp=A[abs(j+1)];A[abs(j+1)]=A[abs(j+0)];
    A[abs(j+0)] = temp; } } } }
```

Figure 6: Best evolved Odd-Even parallel sorting.

best complexity of $O(nlogn)$, while it is quadratic ($O(n^2)$) in nature for the remaining problems.

7. CONCLUSION

We have presented the automatic evolution of efficient parallel iterative sorting that showed an improvement of 25.79% in execution time over preliminary attempt [6].

We attained this both by increasing the flexibility in the design of grammars, and fitness evaluation as opposed to the preliminary investigations that only guarantee program correctness. The efficiency comes from these two alterations.

The most interesting contribution is the automatic load balancing that adapts to the experimental hardware environment, with which, the system has further improved the performance of the evolving programs. However, the analysis shows that code growth has negligible effect except for 2 cores. Finally, we noted that the time complexity of the programs is competing with the attempts in literature.

8. REFERENCES

[1] R. Abbott and J. G. B. Parviz. Guided genetic programming. In H. R. Arabnia and E. B. Kozerenko, editors, *Proceedings of the International Conference on Machine Learning; Models, Technologies and Applications*, pages 28–34. CSREA Press, 2003.

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

[3] 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.

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

[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 sorting programs on multi-cores. In A. M. Mora and G. Squillero, editors, *EvoApplications 2015*, volume 9028 of *LNCS*, pages 706–717. Springer, Heidelberg, 2015.

[7] 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.

[8] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D: Nonlinear Phenomena*, 42(1):228–234, 1990.

[9] K. E. Kinnear Jr. Evolving a sort: Lessons in genetic programming. In *IEEE International Conference on Neural Networks*, pages 881–888. IEEE, 1993.

[10] K. E. Kinnear Jr. Generality and difficulty in genetic programming: Evolving a sort. In S. Forrest, editor, *Proceedings of International Conference on Genetic Algorithms*, pages 287–294. Morgan Kaufmann, 1993.

[11] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[12] 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.

[13] M. O'Neill, M. Nicolau, and A. Agapitos. Experiments in program synthesis with grammatical evolution: A focus on integer sorting. In *IEEE Congress on Evolutionary Computation*, pages 1504–1511, 2014.

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

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

[16] U.-M. O'Reilly and F. Oppacher. An experimental perspective on genetic programming. In R. Manner and B. Manderick, editors, *Parallel Problem Solving from Nature 2*, pages 331–340. Elsevier Science, 1992.

[17] U.-M. O'Reilly and F. Oppacher. A comparative analysis of genetic programming. In P. J. Angeline et. al, editor, *Advances in Genetic Programming 2*, chapter 2, pages 23–44. MIT Press, 1996.

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

[19] C. Ryan, J. J. Collins, and M. O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In W. Banzhaf et. al., editor, *EuroGP 1998*, volume 1391 of *LNCS*, pages 83–95. Springer, Heidelberg, 1998.

[20] C. Ryan and L. Ivan. Automatic parallelization of arbitrary programs. In R. Poli et. al., editor, *EuroGP 1999*, volume 1598 of *LNCS*, pages 244–254. Springer, Heidelberg, 1999.

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

[22] 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.

[23] 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.

[24] P. A. Whigham. *Grammatical Bias for Evolutionary Learning*. PhD thesis, University of New South Wales, 1996.

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