Parallel Genetic Algorithm for Optimizing Compiler Sequences Ordering

International Conference on New Trends in Information and Communications Technology Applications

NTICT 2020: New Trends in Information and Communications Technology Applications pp 128-138 | Cite as

- Manal H. Almohammed (1)
- Ahmed B. M. Fanfakh (1) Email author (ahmed.badri@uobabylon.edu.iq)
- Esraa H. Alwan (1)

1. Department of Computer Science, College of Science for Women, University of Babylon, , Hillah, Iraq

Conference paper
First Online: 13 August 2020

Part of the Communications in Computer and Information Science book series (CCIS, volume 1183)

Abstract

Recent compilers provided many of optimization passes, thus even for expert programmers it is really hard to know which compiler optimization among several optimizations passes can improve program performance. Moreover, the search space for optimization sequences is very huge. Therefore, finding an optimal sequence for even a simple code is not an easy task. The goal of this paper is to find a set of a good optimization sequences using parallel genetic algorithm. The method firstly classifies the programs into three clusters then applying three versions of genetic algorithms each one to cluster in parallel. In order to enhance the result, the migration strategy between these three algorithms is applied. Three optimal sequences at the same time are obtained from this method. However, the proposed method improved the execution time on average by 87% compared with the O2 optimization flag. This method also outperforms the sequential version of genetic algorithm on average of the execution time by **74.8%** in case of using Tournament selection and **72.5%** in case of multi-selection method. LLVM framework is used to validate and execute the proposed method. In addition, Polybench, Standerford, Shootout benchmarks are used as case study to verify the effectiveness of the proposed method.

Keywords

Phase ordering Compiler optimization Parallel genetic algorithm Performance

1 Introduction

Recentcompilers introduce a large number of optimizations such as elimination dead code, replacing certain methods/functions with superior versions, reordering code block, numbering global value, factor, etc. In particular, the order of applying these optimizations can affects the form of code for subsequent optimization phases. However, any optimization applied to the code may improve one or more of the performance metrics such as power, code size, and execution time. Some times when using these optimizations for the program, these optimizations interact with each other in a complex way. This interaction can affect program performance or may produce bad or incorrect code. This problem is known as the phase ordering problem [1]. Therefore, standard compilers introduced optimization levels that are applied for a fixed order, or in some pre-set order that seems to produce comparatively good results. However, many authors have shown that the performance of the application depends mainly on the choice of optimizations by selecting the best order. In the literature, each group of passes with certain order called sequence of passes. Therefore, selecting automatically the best ordering of compiler optimization for any program is difficult task and has significant importance in field of the compiler for a long time [2].

Many researchers have been proposed heuristics and meta-heuristics methods to solve the problem of passes ordering see [3] and [4]. To solve this problem, we proposed in this paper a method to find global optimization sequences of passes and be comprehensive for all programs that give free of interaction and dependency. In our previous work [5], the proposed method uses performance counters to classify the benchmark to three classes. Then, genetic algorithm was executed on each class sequentially. Where each algorithm works independently from each other. However, in this work parallel genetic algorithms are proposed by applying migration strategy to improve the obtained results. This paper is organized as follows: the next Sect. (2) describes how to extract performance events from the performance counter. Section (3) describes the proposed method where the implementation of parallel Genetic Algorithm (PGA) has been explained. Section (4) presents the experimental results obtained from the application of the proposed method. Finally, in Sect. (5), some of the related works have been introduced and the last section, Sect. (6), displays the conclusions and future work.

2 Related Work

T. Satish Kumar et al. present parallel Genetic algorithm that selected the compiler flags to optimize code for multicore CPUs. To compare and test the performance, three methods had been adopted. The first methods compiled the benchmark without applying optimization. The second used randomly selected optimization flags and finally, the compiler optimization levels had been set using parallel genetic algorithm [7]. Pan and Eigenmann [8] introduce a new algorithm called Combined Elimination (CE) which is able to tune the program performance by picking the optimizations passes that can improve the program performance. It can get better or comparable performance with less time to tune the optimization sequence. Kulk arni et al. [9] proposed a comprehensive search strategy to find the best optimization sequences for each of the functions in a program. The relationship between different phases is calculated automatically by doing a comprehensive analytic to these phases. According to these analytical results the execution time is reduced. Cooper, Schielke, and Subramanian [10] used a method to minimize the code size. They used a genetic_algorithm_based approach to find best sequences. Triantafyllis et al. [11] proposed an iterative compilation which is applying various optimization configuration on the same program code segment. Then a comparison among different versions of optimized code segment has occurred and the best one will be chosen. In [12] two complementary general methods described to reduce the search time for finding the optimization sequence using genetic algorithm (GA). The search time is reduced in the first method by avoiding the unnecessary execution of the application. While in second method the same results achieved by modifying the search using a fewer number of generations. The proposed method mainly different from other methods by applying the migration strategy synchronously in parallel.

3 Feature Extraction Using Performance Counter

The information extracted from the performance counter called performance event, which represents the dynamic behavior of the programs. These events consider an important aspect of program improvement, for example, instructions, cache_references, emulation_faults, and others [6] and [7]. On the other hand, these events are an event-oriented portability tool, which can be used to help in solving forward improvement and troubleshooting service [2]. The use of performance counters is attractive because they do not limit the program class, which the system is able to handle. As a result, our system (strategy) can find a good compiler optimization sequence for any program.

4 Proposed Method

In this paper, parallel genetic algorithms are proposed to work on each cluster. Each one used different genetic selection method depending on the offline decision and experiments. Each selection method decided to each cluster by running the genetic algorithm three times one for each cluster. However, for the three different clusters there are three different versions of genetic algorithms that used different selection methods. Moreover, these three genetic algorithms are executed synchronously in parallel, each

one executed over different core of the CPU. Thus, multi-core architecture of the CPU is used to run the three algorithms in parallel using message passing interface (MPI) under C language. MPI is the widest parallel library that works under C or Fortran [13, 14]. Migration strategy is applied to all parallel system by sending the best solutions between the algorithms after some iterations. This strategy used to improve the results by increasing the search space diversity. The resultant of the best sequence for each cluster can be considered as an optimization sequence for any unseen program similar to the features of that cluster.

Table 1 presents the features used in our approach. The first column lists the perf events; the second column gives the used type.

Table 1.

The used features in the proposed approach

Event	Type
Cpu-cycles or cycles, instructions, Cashe- references, Cashe-misses, Bus_cycles	Hardware event
Cpu_clock(msec), Task_clock(msec), Page- faults OR faults, Context-switches, Cpumigrations, Alignment-faults, Emulationfaults	Software event
L1-dcashe-loads, L1-dcashe-loads misses, L1-dcashe-stores, L1-dcashe-storesmisses, L1-icashe-loads, L1-icashe-loadsmisses, L1-icashe-loads, L1-icashe-loads misses, L1-icashe- prefetches, L1-icasheprefetches –misses, LLC-load, LLC-loadsmisses, LLC-strores, LLC-strores misses, LLC-prefetch-misses, Dtlb-loads, Dtlb-loadsmisses, Dtlb-store, Dtlb-store-misses, Dtlb prefetches, Dtlb-prefetches -misses, Itlb -loads, Itlb –loads -misses, Branch-loads, Branch-loads-misses	Hardware cache event
Sched:sched-stat-runtime, Sched:sched-pi-setprio, Syscalls:sys_enter_socket, Kvmmmu:kvm_mmu_pagetable_walk	Tracepoint event
Stalled_cycles-frontend, Stalled_cycle- backend	Dynamic event
Sched:sched_process_exec, Sched:sched_process_frok, Sched:sched_process_wait,Sched:sched_process_wait_task, Sched:sched_process_exit	Tracepoint event

The following is the outline of the proposed method:

- 1-Extracting performance events by using compiler -O0.
- 2-Computed the similarity for each program used in this method as follows:
- $$\mathrm{Sim(p, pi)} = \frac{\sum_{w=1}^{m} Pw \times Piw}{\sqrt{\sum_{w=1}^{m} (Pw)^2}\sqrt{\sum_{w=1}^{m} (Piw)^2}}$$
- where *p* and *pi* represent the base program and other programs respectively [15].
- 3-The programs are divided into three clusters according to them similarity results.
- 4-Running the three genetic algorithms over each cluster synchronously in parallel.
- 5-Iteratively, migration is applied after a predefined number of generations between the algorithms during parallel execution.
- 6-Determining the best sequence for each cluster after finishing the execution of all genetic algorithms.
- 7-For any new unseen program, its similarity is computed by extracting its features and comparing them with the features of all clusters.
- 8-Then, the new unseen program can be matched with the most similar cluster and the optimization sequence of that cluster can be used to optimize the unseen program. Figure 1 illustrate the structure of the proposed method.

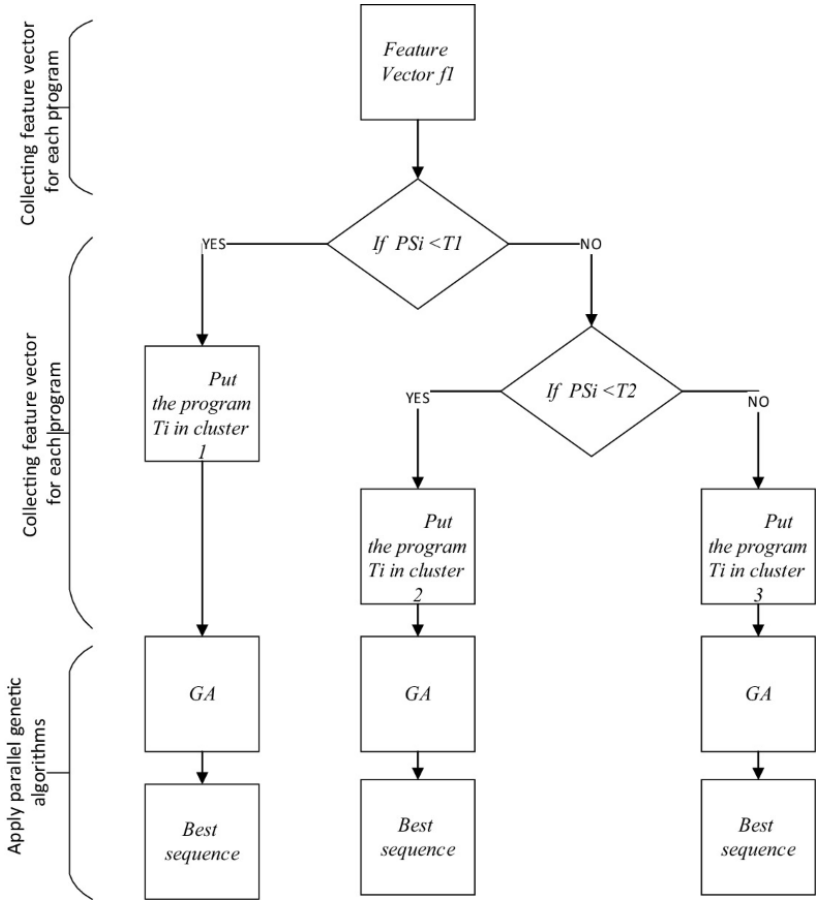

Fig. 1.

The structure of the proposed method

More details about the proposed method are presented as follows:

Step 1: Extracting the Features of the Program

In this method, 60 programs are used where each program collected 52 events. For getting more accuracy, each program executed three times and the average is computed for 52 events. Therefore, the similarity of each program's features with the features of other programs is computed. Depending on the resultant similarity, clustering methods can be applied, for more details about this step see [5].

Step 2: Parallel Genetic Algorithm for Sequence Optimization (PGA)

Genetic algorithm is a metaheuristic search method used to optimize the candidate solutions based on the Darwin evolution principles [6] and [7].In this work, the genetic algorithm is applied to the problem of optimizing sequence ordering. The diversity of the solutions in the genetic algorithm depends mainly on many aspects such as selection,

crossover, mutation methods, and them rates used. One of the most important factors to increase the diversity of the solution in the search space of the genetic algorithm is the migration strategy. Its main idea has subdivided the population of the algorithm into multiple populations and applying migration of the best solution of each population with other ones. The major drawback of this strategy is the high computational cost. To tackle this problem and increasing the diversity of the solutions, three genetic algorithms applying migration strategy are applied synchronously in parallel. Each algorithm, works on specific cluster using different selection operator. Offline experiments are executed to decides the best selection method for each cluster depending on its feature. The selection method in the first algorithm was stochastic universal while in the second algorithm used Rowlett wheel and tournament selection used in the third algorithm. Algorithm (1) is describe the general framework of the three algorithms.

The algorithm used integer representation for the generated chromosomes by maximum length of 64 genes. Each gene can represent a number that may match a pass. To generate variable length chromosomes, the zero-integer value is used to represent the no-pass state. Whereas, the others genes can take an integer number from 1 to 64 which corresponding a specific pass.

Algorithm (1): Parallel Genetic algorithm for sequence optimization
Require:
Pop_size: The population size of the algorithm in cluster.

Opt_passes: The vector of the optimization passes.
Chrom_length: The length of the chromosome in the algorithm.
Pn:Number of the program in the cluster.
Iterations: is the iteration index of the algorithm.
Maxgen: is the maximum number of iterations.
Fit: Fitness value.
Output: *best optimization sequence of passes for selected cluster of programs.*
1: Generating the initial population.
*3: **for** i=1 to pop_size do*
*4: **for** j=1 to chrom_length do*
5: pop [i].sequence[j]⇒select randomly a pass from vector Opt_passes
6: Flip ⇒select random number from 0 to 8
*7: **If** flip=1 then*
8: pop [i].sequence[j]=0
*9: **else***
10: Pop [i].sequence[j]=select random pass from vector Opt_passes.
*11: **endif**.*
12: end for.
13: Compute the average of execution time for the sequence Pop [i]. sequence[j] during three runs applied over Pn programs.
14: Pop1[i].fit = average of the execution time for the three runs.
15: end for.
*16: **repeat***
*Iterations =Iterations **+1***
17: Selecting a chromosome using a one of the selection methods.
18: Applying the uniform crossover over the selected parents.
19: Applying one-point mutation over the two new offspring.
20: Evaluating the new offspring using steps 13 and 14.
21: Replace the best individual from the new produces offspring instead of the worse individual of a randomly selected group.
22: ***Until** the standard deviation between the last two generations reaching to the less fitness function **or Iterations***>maxgen.

Step 3: Application of the Migration Between the Three Algorithms

Three parallel genetic algorithms are executed synchronously in parallel over real four-core processor. Three cores are selected to run the three algorithms in parallel using message passing primitives. Moreover, point-to-point communication functionsare used to apply the migration model. Figure 2 shows the migration model between the three parallel algorithms.

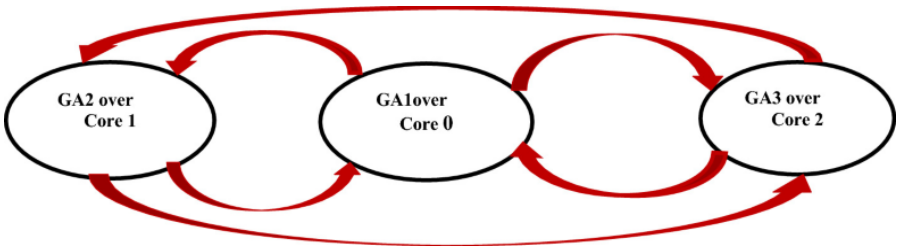

Fig. 2.

The migration model

MPI_Send() and MPI_Recv communication primitives of MPI are used to migrate the selected individuals between the algorithms during the execution time of the parallel system. After a specific number of iterations, migration is applied between the algorithms.

At each algorithm, three individuals are selected randomly and the best one is the migrated individual. Each received individual from the migration is replaced with worst individual selected from a random group of five individuals if its fitness is better from the later individual.

5 Experiments

5.1 The Experimental Setup

This work uses LLVM compiler infrastructure and Linux perf tool to validate the proposed method. Message passing library MPI v3.0.2 has been used to program and execute the parallel algorithms. The LLVM Clang (c language frontend) is used to transform the c source code of each program into IR code which is saved in bitcode machine-readable file format (.bc). O2 optimization level is used to measure the effectiveness of the resulted sequences. Next, following in this section a discusses of the results that is obtained from applying the proposed method to programs selected from three benchmarks and comperes it with the previous work [5].

The proposed method uses a collection of 64 LLVM optimizations passes to find a sequence that will give the best or close to optimal execution time for each cluster. In the proposed method, three populations have been initialized, each one uses a population size equal to 100, the probability of crossover is 0.5, the mutation is 0.01, the ratio of occurrence gene 0.4, the standard deviation for the stopping criteria is 0.01. Moreover, the migration is applied after 10 generations. The maximum chromosome length used in all algorithms is equal to 64 genes.

5.2 Experimental Results

This section presents the results of the proposed parallel genetic algorithms for optimizing the complier sequence ordering problem. To verify the accuracy and the achievement of the proposed parallel framework, two genetic algorithms that used two different strategies are compared with the proposed method. The first algorithm is used tournament selection, uniform crossover and one-point mutation for all clusters. The algorithm executed three times to obtained the results, we refer to this algorithm TSGA. The second algorithm different from the first one only in the selection method used. This algorithm used different selection methods, where each one is selected to fit the feature of each cluster. The Stochastic universal sampling, Roulette wheel, and Tournament selections methods are used for clusters 1,2,3 respectively. Thus, we refer to this algorithm as MSGA. Figures (3, 4, 5) show the comparison results of the proposed parallel genetic algorithm (PGA) with TSGA and MSGA. Figures also presents the comparison of the three algorithms with the standard optimization flag -O2.

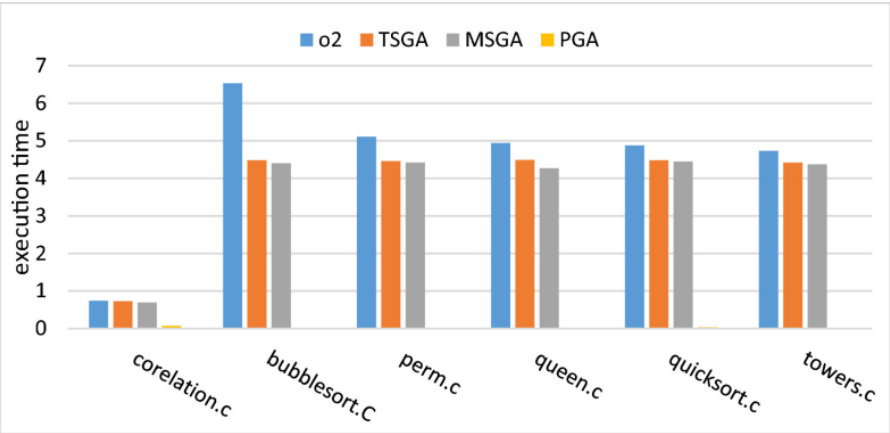

Fig. 3.

Comparison of the execution time between TSGA, MSGA, PGA and the O2 in the first cluster.

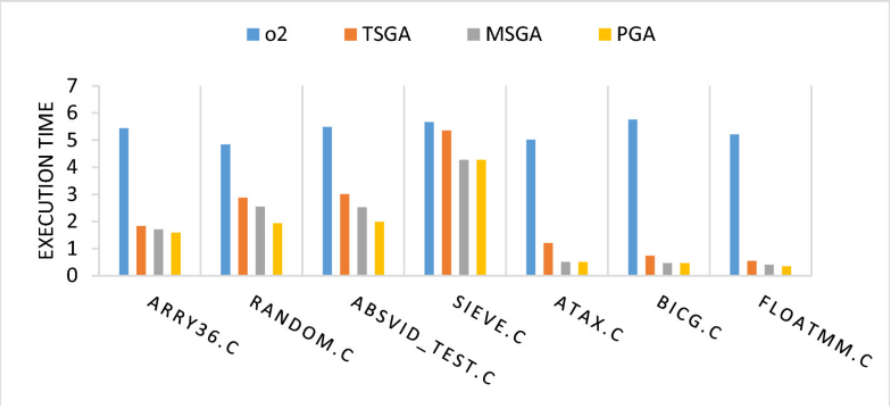

Fig. 4.

Comparison of the execution time between TSGA, MSGA, PGA and the O2 in the second cluster.

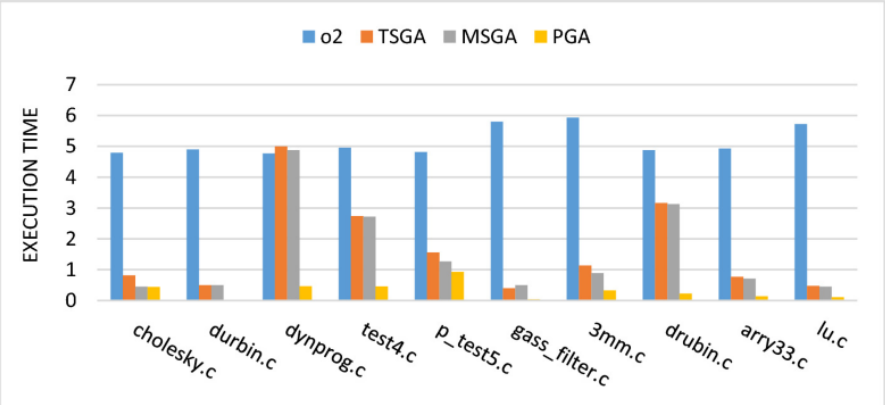

Fig. 5.

Illustrates the comparison of the execution time between TSGA, MSGA, PGA, and the O2 in the third cluster.

The obtained results show that the proposed method gives better results, less execution time, compared to all other methods. This due to the high diversity that introduced from applying the proposed migration strategy in parallel. Moreover, the proposed parallel version reduces the execution time compared to the MSGA algorithm by 3.7. This ratio is the speedup factor that computed as the ration of the sequential execution time divided by the parallel one.

The average of the execution time for all the four methods are computed and presents in the Table 2. The best optimization sequence resulted from each algorithm is presented in Table 3.

Table 2.

The comparison of the execution time between O2,TSGA, MSGA and PGA

Cluster name	O2	TSGA	MSGA	PGA
Cluster 1	4.49	3.84	3.77	0.02
Cluster 2	5.35	2.23	1.78	1.60
Cluster 3	5.15	1.66	1.55	0.32

Table 3.

Best optimization sequence of the second scenario for each cluster

Cluster No.	Best optimization sequence
Cluster 1	-domtree -loop-reduce -mergereturn -sink -gvn -lcssa -loop-simplify -licm -loop-rotate -gvn -instcombine -lcssa -indvars -lowerinvoke -simplifycfg -loop-unroll -indvars -lazy-value-info -dse -loop-instsimplify -targetlibinfo -basiccg -memdep -sccp -globalopt -indvars -simplifycfg -ipconstprop -gvn -scalar-evolution -argpromotion -loop-unswitch -lcssa -early-cse -domtree -lowerswitch -loop-deletion -tailcallelim -early-cse -lower-expect -indvars -loop-rotate -gvn -partial-inliner -reassociate -loop-unroll -lowerswitch -dse -loop-simplify -loops -inline -constprop -loop-simplify -loop-instsimplify -strip-dead-prototypes -globaldce
Cluster 2	-inline -lowerinvoke -scalar-evolution -tailcallelim -gvn -loop-rotate -indvars -codegenprepare -inline -mergereturn -ipconstprop -codegenprepare -mergereturn -targetlibinfo -loop-instsimplify -argpromotion -instcombine -constmerge -mergefunc -lowerinvoke -lower-expect -constprop -loop-unswitch -licm -instsimplify -tailcallelim -partial-inliner -constprop -lcssa -reassociate -basiccg -lowerswitch -basicaa -loop-instsimplify -instsimplify -simplifycfg -lcssa -mergereturn -scalar-evolution -inline -deadargelim -loop-simplify -strip-dead-prototypes -codegenprepare -loops -instcombine -constprop -dse -lowerswitch -lowerinvoke -simplifycfg
Cluster 3	-loweratomic -constmerge -gvn -inline -loop-idiom -globaldce -memcpyopt -constmerge -dce -deadargelim -lower-expect -indvars -gvn -instcombine -memdep -gvn -ipsccp -prune-eh -lowerswitch -licm -ipconstprop -loop-idiom -early-cse -reassociate -constmerge -domtree -simplifycfg -inline -lazy-value-info -inline -memcpyopt -instsimplify -deadargelim -argpromotion -strip-dead-prototypes -tbaa -basiccg -simplifycfg -functionattrs -tbaa -lcssa -loop-simplify -mergereturn -loop-instsimplify -argpromotion -codegenprepare -memcpyopt -loop-deletion -loop-reduce -early-cse -adce -correlated-propagation -simplifycfg

6 Conclusion

This paper described using parallel genetic algorithm to discover the best optimization sequence. Each genetic algorithm runs over one core. Migration was applied between the populations. The proposed method gave three optimal sequences at the same time. Moreover, the proposed method compared with two different sequential versions of genetic algorithm. The comparison results showed that the proposed parallel version outperforms the other two version of the genetic algorithm due to migration strategy used in parallel.

LLVM infrastructure has been used to validate the proposed method. The experiment results obtained indicate the effectiveness of the proposed method when it compared with the earlier work [5]. Moreover, each obtained sequence for a cluster of programs can be used as a guided sequence for the new unseen program. The similarity was computed between the features of the unseen program with features of the each cluster. Therefor the sequence of the most similar cluster is used to optimize the unseen program. In the future, more programs can be used to expand the number of clusters. Thus, the accuracy of the computed similarity between programs' clusters and the unseen program can be increased. Moreover, the parallel implementation of the genetic algorithm will expand to consider the other genetic operators such as crossover and mutation.

References

1. Sher, G., Martin, K., Dechev, D.: Preliminary results for neuroevolutionary optimization phase order generation for static compilation. In: Proceedings of the 11th Workshop on Optimizations for DSP and Embedded Systems, pp. 33–40 (2014)
Google Scholar (https://scholar.google.com/scholar?q=Sher%2C%20G.%2C%20Martin%2C%20K.%2C%20Dechev%2C%20D.%3A%20Preliminary%20results%20for%20neuroevolutionary%20optimization%20phase%20order%20generation%20for%20static%20compilation.%20In%3A%20Proceedings%20of%20the%2011th%20Workshop%20on%20Optimizations%20for%20DSP%20and%20Embedded%20Systems%2C%20pp.%2033%E2%80%9340%20%282014%29)

2. Alkaaby, Z.S., Alwan, E.H., Fanfakh, A.B.M.: Finding a good global sequence using multi-level genetic algorithm. J. Eng. Appl. Sci. 9777–9783 (2018)
Google Scholar (https://scholar.google.com/scholar?q=Alkaaby%2C%20Z.S.%2C%20Alwan%2C%20E.H.%2C%20Fanfakh%2C%20A.B.M.%3A%20Finding%20a%20good%20global%20sequence%20using%20multi-level%20genetic%20algorithm.%20J.%20Eng.%20Appl.%20Sci.%209777%E2%80%939783%20%282018%29)

3. Crainic, T.G., Toulouse, M.: Parallel strategies for meta-heuristics. In: Glover, F., Kochenberger, G.A. (eds.) Handbook of Metaheuristics. International Series in Operations Research & Management Science, vol. 57, pp. 475–513. Springer, Boston (2003). https://doi.org/10.1007/0-306-48056-5_17 (https://doi.org/10.1007/0-306-48056-5_17)

4. Hertz, A., Widmer, M.: Guidelines for the use of meta-heuristics in combinatorial optimization. Eur. J. Oper. Res. **151**, 247–252 (2003)
MathSciNet (http://www.ams.org/mathscinet-getitem?mr=2014074)
CrossRef (https://doi.org/10.1016/S0377-2217(02)00823-8)
Google Scholar (http://scholar.google.com/scholar_lookup?title=Guidelines%20for%20the%20use%20of%20meta-heuristics%20in%20combinatorial%20optimization&author=A.%20Hertz&author=M.%20Widmer&journal=Eur.%20J.%20Oper.%20Res.&volume=151&pages=247-252&publication_year=2003)

5. Almohammed, M.H., Alwan, E.H., Fanfakh, A.B.M.: Programs features clustering to find optimization sequence using genetic algorithm. In: Jain, L.C., Peng, S.-L., Alhadidi, B., Pal, S. (eds.) ICICCT 2019. LAIS, vol. 9, pp. 40–50. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-38501-9_4 (https://doi.org/10.1007/978-3-030-38501-9_4)
CrossRef (https://doi.org/10.1007/978-3-030-38501-9_4)
Google Scholar (http://scholar.google.com/scholar_lookup?title=Programs%20features%20clustering%20to%20find%20optimization%20sequence%20using%20genetic%20algorithm&author=MH.%20Almohammed&author=EH.%20Alwan&author=ABM.%20Fanfakh&pages=40-50&publication_year=2020)

6. Nisbet, A.P.: GAPS: Iterative feedback directed parallelization using genetic algorithms. In: Workshop on Profile and Feedback-Directed Compilation (1998)
Google Scholar (https://scholar.google.com/scholar?q=Nisbet%2C%20A.P.%3A%20GAPS%3A%20Iterative%20feedback%20directed%20parallelization%20using%20genetic%20algorithms.%20In%3A%20Workshop%20on%20Profile%20and%20Feedback-Directed%20Compilation%20%281998%29)

7. Kumar, T.S., Sakthivel, S., Kumar, S.: Optimizing code by selecting compiler flags using parallel genetic algorithm on multicore CPUs. Int. J. Eng. Technol. (IJET) **6**, 544–555 (2014)
Google Scholar (http://scholar.google.com/scholar_lookup?title=Optimizing%20code%20by%20selecting%20compiler%20flags%20using%20parallel%20genetic%20algorithm%20on%20multicore%20CPUs&author=TS.%20Kumar&author=S.%20Sakthivel&author=S.%20Kumar&journal=Int.%20J.%20Eng.%20Technol.%20%28IJET%29&volume=6&pages=544-555&publication_year=2014)

8. Pan, Z., Eigenmann, R.: Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 319–332. IEEE Computer Society (2006)
Google Scholar (https://scholar.google.com/scholar?q=Pan%2C%20Z.%2C%20Eigenmann%2C%20R.%3A%20Fast%20and%20effective%20orchestration%20of%20compiler%20optimizations%20for%20automatic%20performance%20tuning.%20In%3A%20Proceedings%20of%20the%20International%20Symposium%20on%20Code%20Generation%20and%20Optimization%2C%20pp.%20319%E2%80%93332.%20IEEE%20Computer%20Society%20%282006%29)

9. Kulkarni, P.A., Whalley, D.B., Tyson, G.S., Davidson, J.W.: Exhaustive optimization phase order space exploration. In: Proceedings of the International Symposium on Code Generation and Optimization (2006)
Google Scholar (https://scholar.google.com/scholar?q=Kulkarni%2C%20P.A.%2C%20Whalley%2C%20D.B.%2C%20Tyson%2C%20G.S.%2C%20Davidson%2C%20J.W.%3A%20Exhaustive%20optimization%20phase%20order%20space%20exploration.%20In%3A%20Proceedings%20of%20the%20International%20Symposium%20on%20Code%20Generation%20and%20Optimization%20%282006%29)

10. Cooper, K.D., Schielke, P.J., Subramanian, D.: Optimizing for reduced code space using genetic algorithms. In: ACM SIGPLAN Notices, pp. 1–9 (1999)
Google Scholar (https://scholar.google.com/scholar?q=Cooper%2C%20K.D.%2C%20Schielke%2C%20P.J.%2C%20Subramanian%2C%20D.%3A%20Optimizing%20for%20reduced%20code%20space%20using%20genetic%20algorithms.%20In%3A%20ACM%20SIGPLAN%20Notices%2C%20pp.%201%E2%80%939%20%281999%29)

11. Triantafyllis, S., Vachharajani, M., Vachharajani, N., August, D.I.: Compiler optimization-space exploration. In: Proceedings of the International Symposium on Code Generation and Optimization. Feedback-Directed and Runtime Optimization, 204–215, March 2003
Google Scholar (https://scholar.google.com/scholar?q=Triantafyllis%2C%20S.%2C%20Vachharajani%2C%20M.%2C%20Vachharajani%2C%20N.%2C%20August%2C%20D.I.%3A%20Compiler%20optimization-space%20exploration.%20In%3A%20Proceedings%20of%20the%20International%20Symposium%20on%20Code%20Generation%20and%20Optimization.%20Feedback-Directed%20and%20Runtime%20Optimization%2C%20204%E2%80%93215%2C%20March%202003)

12. Kulkarni, P.A., Hines, S.R., Whalley, D.B., Hiser, J.D., Davidson, J.W., Jones, D.L.: Fast and efficient searches for effective optimization-phase sequences. ACM Trans. Archit. Code Optim. **2**, 165–198 (2005)
CrossRef (https://doi.org/10.1145/1071604.1071607)
Google Scholar (http://scholar.google.com/scholar_lookup?title=Fast%20and%20efficient%20searches%20for%20effective%20optimization-phase%20sequences&author=PA.%20Kulkarni&author=SR.%20Hines&author=DB.%20Whalley&author=JD.%20Hiser&author=JW.%20Davidson&author=DL.%20Jones&journal=ACM%20Trans.%20Archit.%20Code%20Optim.&volume=2&pages=165-198&publication_year=2005)

13. Guiffaut, C., Mahdjoubi, K.: A parallel FDTD algorithm using the MPI library. IEEE Antennas Propag. Mag. **43**, 94–103 (2001)
CrossRef (https://doi.org/10.1109/74.924608)
Google Scholar (http://scholar.google.com/scholar_lookup?title=A%20parallel%20FDTD%20algorithm%20using%20the%20MPI%20library&author=C.%20Guiffaut&author=K.%20Mahdjoubi&journal=IEEE%20Antennas%20Propag.%20Mag.&volume=43&pages=94-103&publication_year=2001)

14. Idrees, S.K., Fanfakh, A.B.M.: Performance and energy consumption prediction of randomly selected nodes in heterogeneous cluster. In: Al-mamory, S.O., Alwan, J.K., Hussein, A.D. (eds.) NTICT 2018. CCIS, vol. 938, pp. 21–34. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01653-1_2 (https://doi.org/10.1007/978-3-030-01653-1_2)
CrossRef (https://doi.org/10.1007/978-3-030-01653-1_2)
Google Scholar (http://scholar.google.com/scholar_lookup?title=Performance%20and%20energy%20consumption%20prediction%20of%20randomly%20selected%20nodes%20in%20heterogeneous%20cluster&author=SK.%20Idrees&author=ABM.%20Fanfakh&pages=21-34&publication_year=2018)

15. Ashouri, A.H., Bignoli, A., Palermo, G., Silvano, C., Kulkarni, S., Cavazos, J.: MiCOMP: mitigating the compiler phase-ordering problem using optimization

sub-sequences and machine learning. ACM Trans. Archit. Code Optim. (TACO) **14**(3), 29 (2017)
Google Scholar (http://scholar.google.com/scholar_lookup?title=MiCOMP%3A%20mitigating%20the%20compiler%20phase-ordering%20problem%20using%20optimization%20sub-sequences%20and%20machine%20learning&author=AH.%20Ashouri&author=A.%20Bignoli&author=G.%20Palermo&author=C.%20Silvano&author=S.%20Kulkarni&author=J.%20Cavazos&journal=ACM%20Trans.%20Archit.%20Code%20Optim.%20%28TACO%29&volume=14&issue=3&pages=29&publication_year=2017)

Copyright information

About this paper

Cite this paper as:

Almohammed M.H., Fanfakh A.B.M., Alwan E.H. (2020) Parallel Genetic Algorithm for Optimizing Compiler Sequences Ordering. In: Al-Bakry A. et al. (eds) New Trends in Information and Communications Technology Applications. NTICT 2020. Communications in Computer and Information Science, vol 1183. Springer, Cham. https://doi.org/10.1007/978-3-030-55340-1_9

- First Online 13 August 2020
- DOI https://doi.org/10.1007/978-3-030-55340-1_9
- Publisher Name Springer, Cham
- Print ISBN 978-3-030-55339-5
- Online ISBN 978-3-030-55340-1
- eBook Packages Computer Science Computer Science (R0)

Personalised recommendations