Why Asynchronous Parallel Evolution is the Future of Hyper-heuristics: A CDCL SAT Solver Case Study

Alex R. Bertels
Natural Computation Laboratory
Department of Computer Science
Missouri University of Science and Technology
Rolla, Missouri, U.S.A.
arb9z4@mst.edu

Daniel R. Tauritz
Natural Computation Laboratory
Department of Computer Science
Missouri University of Science and Technology
Rolla, Missouri, U.S.A.
dtauritz@acm.org

ABSTRACT

Evolutionary Algorithms (EAs) are inherently parallel due to their ability to simultaneously evaluate the fitness of individuals. Synchronous Parallel EAs (SPEAs) leverage this with the intent to gain significant speed-ups when executed on multiple processors. However, many important problem classes lead to large variations in fitness evaluation times, such as is often the case in hyper-heuristics where the time complexity of executing one individual may differ greatly from that of another. Asynchronous Parallel EAs (APEAs) omit the generational synchronization step of traditional EAs which work in well-defined cycles. They can provide scalability improvements proportional to the variation in fitness evaluation times of the evolved individuals, and therefore should be considered for use in hyper-heuristics. This paper provides an empirical analysis of the improvements obtained by applying APEAs, compared to SPEAs, on a case study involving the evolution of conflict-driven clause learning Boolean satisfiability solvers, demonstrating that APEAs are the future of hyper-heuristics.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search; I.2.2 [**Artificial Intelligence**]: Automatic Programming–*program modification, program synthesis*

Keywords

Hyper-Heuristics, Asynchronous Parallel Evolution, Parallel Evolutionary Algorithms, Genetic Programming, SAT

1. INTRODUCTION

The computational time needed to evaluate the fitness of a trial solution in hyper-heuristics for many important problem classes is characterized by large variations in the computational time needed to evaluate the fitness of a trial solution. One example is when dealing with sampled fitness, where fitness may for instance be estimated by applying the trial solution to a set of instances sampled from a problem domain specific distribution; some instances may take significantly more computational time to evaluate than others. Another example is when the complexity of the genotypes in a population can significantly vary, such as is often the case in Genetic Programming (GP), where typically limits are placed on genotype size (tree depth in Koza style GP) and larger genotypes are penalized to create parsimony pressure to combat bloat [7, 23]. For example, when comparing two complex evolved algorithms where the runtime of each is both dependent on their internal structure as well as the specific sampled problem instances their fitness is being approximated on, the variation in computational time is caused partially by their structural differences (e.g., one algorithm might have an extra loop nesting) and partially by the instance structure differences (e.g., one instance might require full computation over the entire instance while another can be determined with a partial computation).

Hyper-heuristics are a type of meta-heuristic which search program space for the purpose of automating the design of algorithms. They typically employ GP and their fitness evaluation relies on a sample consisting of multiple test cases [21, 20]. Thus, they can suffer from both the sampled fitness and the varying genotype complexity causes of fitness evaluation time variation, which greatly amplifies the time variation. However, if the variation introduced by the genotype complexity is minuscule in comparison to that presented by the context in which the generated heuristics are evaluated, then the time variation may be bounded by the given application.

Consider evolving heuristics in conflict-driven clause learning (CDCL) Boolean satisfiablity (SAT) solvers to target specific problem classes. Encoding a specific problem class in SAT creates a class of structured logical expressions, called SAT instances. The variable interactions or associations in each instance in a class define a distinct structure. Efficiently solving instances in a specific class requires finding the solver and parameter configuration that perform best for that class.

As CDCL SAT solvers attempt to find a solution to an instance or prove that the instance is unsatisfiable, the solver must make decisions for the assignment of Boolean variables in the logical expression. A function – labelled as the variable scoring heuristic – assists in determining which variables are selected as decision variables. Better variable selections often equate to fewer decisions and shorter solving

GECCO'16 Companion, July 20-24, 2016, Denver, CO, USA

DOI: http://dx.doi.org/10.1145/2908961.2931729

times. The variable scoring heuristic can be represented in GP and requires evaluating against a sample set of SAT instances. Thus, the evaluation times can vary drastically with the difficulty of the datasets, the sample size, and the quality of the heuristic. Additionally, these factors can also result in lengthy evolution times.

Given a distributed computing resource, such as a multi-core machine, or a parallel cluster, Evolutionary Algorithms (EAs) are often able to reduce overall runtime by distributing individuals in the population to be evaluated concurrently. Synchronous Parallel EAs (SPEAs) maintain the generational step that is typical to most EAs; however, this approach suffers from idle CPU cycles when the fitness evaluation times vary. Asynchronous Parallel EAs (APEAs) eliminate this wasted time by producing offspring as slave nodes in the distributed computing resource become idle. This paper empirically studies the relative performance of APEAs versus SPEAs on global populations of CDCL SAT solver variable scoring heuristics, as opposed to distributed populations such as in island-model or diffusion EAs [2]. The rest of the paper is organized as follows. Section 2 reviews related work on Parallel Evolutionary Algorithms (PEAs) and evolving heuristics for SAT solvers. Section 3 illustrates the structure of the hyper-heuristic and its mechanics. This is followed by a description of the experimental setup in Section 4 and results in Section 5. Lastly, the conclusions and future work are discussed in Section 6 and Section 7, respectively.

2. RELATED WORK

2.1 Parallel Evolutionary Algorithms

Populations within a PEA can either be structured as a single, centralized population or as multiple decentralized subpopulations [24]. Distributing subpopulations over available machines achieves near-linear scalability, with the sole overhead due to inter-population communication through interchange of select individuals at typically fixed time intervals called epochs. This allows for each subpopulation to evolve semi-independently, while slowly diffusing genetic material throughout all populations. Alba et al. [1, 2, 3] have reported on the effectiveness of various behaviors, particularly distributed and cellular reproduction, in distributed PEAs.

Durillo et al. [9] have shown empirical evidence supporting the significant improvement in terms of various quality metrics when employing APEAs rather than SPEAs for NSGA-II. The APEA master process creates and sends individuals to be evaluated as the slave processors become idle. In the generational version, the population is replaced when enough offspring have been generated. With the steady-state alternative, the offspring are considered as each is received. The researchers employed homogeneous populations as the test cases during experimentation. While these results still apply to heterogeneous populations, an in-depth runtime analysis should be completed to measure performance.

Those that have specifically addressed heterogeneous populations, note that APEAs are biased toward individuals with shorter evaluation times [8, 29, 30]. This is a result of the master process receiving those individuals sooner and more often, flooding the population. This potentially reduces the search space that can be reached within the runtime. Yagoubi and Schoenauer [29] attempted to circumvent this with a duration-based selection on the received offspring. This supposed defect can be taken advantage of in various situations, one of which is evolving genetic programs, which must use a mechanism such as parsimony pressure or must minimize a size-related objective value to prevent any individual from becoming too large. If the size of the individual is proportionate to the evaluation time, then the bias provided by heterogeneous evaluation times can be used to produce an implicit parsimony pressure [28, 19]. This characteristic may not necessarily be present in some hyper-heuristics if the size of the genotype has little effect on the evaluation time, as is noted in the present application.

2.2 SAT Solving

Recent work has automatically optimized SAT solver parameter configuration to target specific classes of structured instances [17, 16, 12], and other work automatically evolved variable selection techniques for stochastic local search (SLS) solvers [4, 18, 13, 14, 15]. However, CDCL solvers are still the most efficient SAT solvers. As a deterministic alternative to SLS solvers, CDCL SAT solvers construct reason clauses to learn from conflicting variable assignments throughout the search. Although Biere and Fröhlich [6, 5] have demonstrated that restart and variable selection schemes drastically impact CDCL solver efficiency for specific problem classes, we know of no work that automatically evolves CDCL heuristics. We believe that appropriate CDCL operations will increase the effectiveness of a CDCL SAT solver in targeting classes of instances with unique structure.

We take the next step by applying hyper-heuristics to generate CDCL SAT solvers tailored for an arbitrary, but particular, problem class. As is typical, our hyper-heuristic – named ADSSEC (**A**utomated **D**esign of **S**AT **S**olvers employing **E**volutionary **C**omputation) – employs GP to automatically reorganize and manipulate the algorithmic primitives constituting the variable scoring heuristic [20, 22]. These primitives can be as general as state-related variables and binary operations or as specific as carefully constructed functions with tunable inputs. The primitives employed are more granular than statements in source code and are therefore much more versatile in developing new solver components than the line substitution approach proposed by Petke et al. designed to optimize entire SAT solvers [26, 25].

3. EVOLUTIONARY APPROACH

Influenced by the success of Fukunaga [13] in improving SLS runtimes by evolving specific heuristics, we use GP to evolve variable scoring heuristics to automatically target CDCL solvers to specific classes of structured instances. In particular, we use Koza-style GP [27] as it is well suited to representing the parse trees of variable scoring heuristics. ADSSEC creates an initial population of variable scoring heuristics and evolves the population through mutation and recombination. ADSSEC evaluates these heuristics by replacing the variable selection heuristic in MiniSat [10], a commonly used efficient and deterministic CDCL solver with dense source code. While ADSSEC employs a standard parent selection before producing offspring, we constructed a survival selection specifically for the APEA. ADSSEC re-

turns the heuristics from the final population after reaching the termination criteria.

3.1 Heuristic Representation

Mapping variable scoring heuristic functions to objects easily manipulated in a GP is fairly straight-forward. We represent each scoring heuristic as a parse tree where non-terminal nodes are operators and terminal nodes are state-related values (see below).
Derived from currently implemented variable scoring heuristics [6], ADSSEC defines the following terminal nodes:

- Score (s): The previous score of the variable.
- Conflict count (i): The current number of conflicts encountered. Also called the conflict index.
- Variable Decay (f): Initially used as the rate of variable score decay in MiniSat. Now f is just used to derive the Variable Increment value (*MiniSat Default: 0.95*).
- Variable Increment ($g = (1/f)^i$): The amount MiniSat increases a variable's score.
- Constant (C): A constant value in $\{1, 2, 3, \ldots, 10\} \cup \{0.0, 0.1, 0.2, \ldots, 0.9\}$.
- Special Component (H): Derived from the Chaff CDCL SAT solver for scaling variable scores [6].

$$h_i^m = \begin{cases} 0.5 \cdot s & \text{if } m \text{ divides } i \text{ evenly} \\ s & \text{otherwise} \end{cases} \quad (1)$$

where m is a power of 2: $\{2, 4, 8, \ldots, 1024\}$.

ADSSEC defines the following non-terminal (binary) operator nodes: Addition ($+$), Subtraction ($-$), Multiplication ($*$), and Division ($/$). These arithmetic operators may be applied because all the terminal nodes relate to either integer or floating-point values.

Again, ADSSEC evolves the parse tree genetic encodings. To evaluate each variable scoring heuristic, we convert the parse tree into a C++ statement. We replace the original variable scoring heuristic in a pre-built MiniSat 2.2 by compiling and linking in the new variable scoring heuristic (C++ statement). We execute the resulting solver on test instances to evaluate the effectiveness of the heuristic. This method allows us to take advantage of MiniSat and the performance derived from being implemented in C++ while reducing development time.

3.2 Objective

The objective score represents how well an evolved version of MiniSat performs on a provided training set of instances. The true objective score is a function (e.g., average) over all instances in the problem class being targeted. However, as it is infeasible to compute over a potentially infinite set of instances, instead a small sampling of instances provides an approximation as is discussed later. Determining the best performance measure to use in ADSSEC is difficult. Traditionally we aim to reduce the runtime needed to either find a solution or prove unsatisfiability. However, because ADSSEC evaluates several instances in parallel on the same hardware, runtimes for individual instances are inconsistent even with a deterministic solver. Therefore, instead of runtime, we use the number of variable decisions as a consistent metric. The number of decisions serves as an approximation of the number of steps needed to solve an instance and is not affected by the load of the machine the solver is executed on. Fewer decisions should indicate fewer steps and, subsequently, lower runtimes; an improved variable scoring heuristic should reduce this value. Our per-instance sub-score for an evolved variant is the ratio of the number of decisions needed by the variant to that needed by the original selection scheme.

The objective score is then simply the average of all the instance sub-scores. Given that all the sub-scores are expressed relative to the original MiniSat, any evolved individual that performs identically to MiniSat's variable scoring heuristic will end up with an objective score of 1.0. Lower scores indicate better schemes.

Occasionally, the EA will construct inadequate heuristics that cause the solver to require an inordinate number of decisions to reach a conclusion. We define limiting functions to prevent wasting evaluation time on such heuristics. ADSSEC relies on default MiniSat's performance to approximate reasonable limits for any given SAT instance. Initially, ADSSEC limits an evolved MiniSat to three times the number of decisions the original MiniSat needed to solve that instance. While the multiplier of three is user-configurable, manual tuning indicated that this limit was fairly generous without wasting an excessive amount of evaluation time. These generous limits are required to collect decent heuristics in the population – decent heuristics provide complex genetic material for later optimization; they do not time out on all tested instances, but are generally worse than the original MiniSat. However, as ADSSEC progresses through the evolutionary process, our interest shifts to exploiting the heuristics that are strictly better than the original. As such, the decision limit linearly decreases down to the exact number of decisions MiniSat needed for a specific instance or the average number of decisions for the sample set. For example, if ADSSEC is to complete 5000 evaluations throughout the run and the decision limit multiplier decreases from 3.0 to 1.0, then the multiplier is decremented by $((3.0 - 1.0)/5000 = 0.0004$ after each evaluation.

Ideally an accurate objective score would be determined by executing the evolved variable scoring heuristic against the entire dataset of interest. Because this is generally too costly, ADSSEC utilizes strike-based sampling to gauge the effectiveness of a MiniSat variant. ADSSEC randomly selects a user-defined number of instances from the given training set to evaluate a variant. At the start of evolution, the population contains mostly low-quality heuristics and evaluating these heuristics against difficult instances wastes evaluation time. As such, a bias is placed in favor of selecting easier instances in early evolution. This bias linearly transforms to a uniform selection at the end of the evolutionary process. For each instance in this selection, ADSSEC executes the evolved variant and assigns a sub-score ratio as described before. If that variant reaches the decision limit for that instance, then the variant receives a strike and a sub-score of the current decision-limit multiplier. If a variant reaches a user-defined number of strikes during evaluation, then the sub-scores for the remaining instances in the sample are assigned the current decision-limit multiplier.

3.3 Evolutionary Algorithm

3.3.1 Population Initialization

To create each individual in the initial population of variable scoring heuristics, ADSSEC randomly generates a parse tree from the primitives, or nodes, described previously (see Sect. 3.1). First, as experimentation shows that no single terminal node produces an effective scoring scheme, ADSSEC assigns a random operator node to the root of the tree. ADSSEC then assigns two random nodes to the left and right branches of the operator node. There is a 50% chance that each node will be terminal (versus non-terminal). If the node is non-terminal, then ADSSEC repeats the process and assigns a random operator node. If the node is terminal, then there is a 50% chance that the node will be the previously assigned score, s; if not s, ADSSEC randomly assigns one of the other terminal node options. We introduced this bias because most current schemes appear to rely on the previous variable score. We manually tuned the maximum depth of a tree generated for the initial population to eight. Smaller depths contained much less genetic diversity, while larger trees produced complex heuristics that rarely solved instances in the decision limits.

3.3.2 Parent Selection and Variation Operators

ADSSEC uses one of two methods to develop a single offspring (variant): mutation or recombination. For mutation, ADSSEC simply randomly selects a subtree in a random individual's parse tree and replaces it with a new branch generated using the rules established in Population Initialization (Sect. 3.3.1) – without a depth limitation. Typically, APEAs mitigate tree growth by promoting an implicit parsimony pressure. This is under the assumption that smaller trees have shorter evaluation times and return to the population sooner. However, the strike-based sampling terminated bad heuristics quickly, which partially eliminated the implicit pressure. Fortunately, most overly-complicated heuristics receive poor objective scores and are removed during survival selection. For recombination, ADSSEC implements a sub-tree crossover: the system randomly selects two individuals in the population and replaces a random branch from the first parent with a random branch from the second parent. Again, this procedure only produces a single child.

3.3.3 Survival Selection

The survival selection chooses which individuals in the population continue into the next generation. In ADSSEC, we want to encourage genetically diverse selection so that smaller parse trees (which are generated more easily) do not flood the population. Certain small heuristics have adequate performance and, had one been discovered early on in evolution, could be spread throughout the population if diversity was not maintained.

Crowding functions are selection functions that excel at promoting genetic diversity in the population [11]. In a deterministic crowding function, an offspring competes with its closest parent, either replacing the parent or being dropped from the population in favor of the parent. In an APEA, however, generations are not clearly delineated and a parent can have multiple offspring being evaluated simultaneously. We developed an asynchronous crowding function that allows offspring to compete with either their parents or any *'siblings'* – or potentially descendants of siblings – that replaced the parents. We use a computationally cheap distance function comparing histograms of node types (e.g., addition, constant, conflicts, etc.) to determine the closest remaining relative in the current population. To provide a fair comparison between the models, the SPEA version of ADSSEC employs the same crowding selection method.

Parents have to be uniformly selected at random to ensure that each individual has an equal chance of providing genetic material to the pool. Additionally, uniform selection allows an equal chance of producing offspring, which can eliminate less fit parents from the population with the employed survival selection.

3.3.4 Termination

ADSSEC terminates the evolutionary cycle after completing a user-defined number of evaluations. However, throughout the run individuals may be replaced by randomly generated parse trees if the population has become stale or has converged. If the best individual has not been improved in a user-defined number of evaluations, ADSSEC introduces new material to the gene pool. Currently, we replace all variants whose performance is worse than that of the original MiniSat. This mechanism is useful in restarting the exploration of the variable scoring heuristics search space.

4. EXPERIMENTATION

Ideally, we want to construct entire solvers for a given problem, and ADSSEC demonstrates the obstacles and, more importantly, the potential of adapting a single component of a CDCL solver. Our experiments with the prototype ADSSEC system require datasets that have:

- instances that ADSSEC can feasibly train on in a short period of time (each instance should require at most a few seconds for the original MiniSat to solve)
- enough instances to sufficiently represent a distinct instance class for both training and testing
- instances that are difficult enough that the instance can benefit from a fitted heuristic

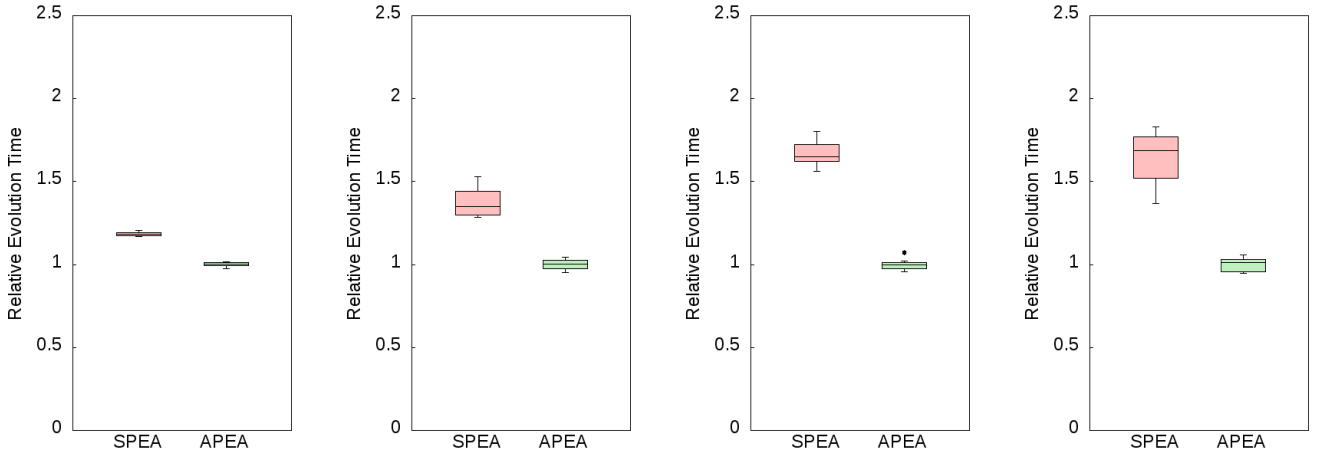
Unfortunately, these requirements make many of the usual SAT datasets inappropriate for our initial prototype experiments. Instances from previous SAT competitions attempt to challenge the capabilities of the solvers, so many require too significant an amount of time to solve. Many publicly available datasets contain too few instances to sufficiently represent the distinct problem class or the instances are so simple that nothing is gained by creating a fitted heuristic. We chose to generate datasets for ADSSEC as generators provide us with enough control to meet these criteria while keeping us from presenting bias by hand-selecting specific instances.

We used a modularity-based generator developed by Jesús Giráldez Cru[1] to create 80 instances for the datasets. These instances simulate an underlying structure that may be found in a class of instances from industry. The generator allows the user to specify the structure of each instance; we generated instances that encapsulated 90 modularity communities – as defined within the context of the tool – and 3 literals per clause. Half of the instances were configured with a modularity of 0.85 and the other 40 with 0.9. We used seeds 1

[1]http://www.iiia.csic.es/~jgiraldez/

Table 1: ADSSEC EA parameter settings

Population (μ)	Offspring (λ)	Mutation Rate	Crossover Rate	Termination Evaluations	Restart Evaluations	Dec. Limit Multiplier	Sample Size
20	20	0.10	0.90	1000	100	$3.0 \rightarrow 1.0$	15 (5 strikes)



(a) Dataset A (Mean: 7,441 Decisions) (Std Dev.: 11,798 Decisions)

(b) Dataset B (Mean: 99,313 Decisions) (Std Dev.: 118,761 Decisions)

(c) Dataset C (Mean: 297,925 Decisions) (Std Dev.: 309,996 Decisions)

(d) Dataset D (Mean: 522,668 Decisions) (Std Dev.: 565,955 Decisions)

Figure 1: Boxplots of evolution time relative to the mean of the evolution time for the asynchronous runs on each respective dataset. The average relative asynchronous evolution time will always be at 1.0 for each plot. The average and standard deviation of the decisions needed by MiniSat to solve the instances used in each dataset describe the difficulty and variation that can be expected to influence evaluation times.

through 40 for both halves. Each instance contained 5000 variables in 19000 clauses. A majority of the instances were satisfiable while 32 were unsatisfiable. We trained ADSSEC on select subsets of 32 instances where the sample size was 15 allowing up to 5 strikes per evaluation. To obtain the subsets, we first sorted the 80 instances by the number of decisions default MiniSat needed to solve each. Then, the instances were divided into five groups of sixteen instances, where Group 0 needed the least number of decisions and Group 4 needed the most decisions. The final subsets used for training were constructed as follows:

Dataset A Combined Group 0 and Group 1

Dataset B Combined Group 0 and Group 2

Dataset C Combined Group 0 and Group 3

Dataset D Combined Group 0 and Group 4

Dataset A has the least variation in evaluation time and Dataset D has the most. Some instances can be solved in less than a thousandth of a second while the longer ones can take several seconds.

The computationally extensive search made automated tuning of ADSSEC's parameters infeasible; thus, we used manual tuning to discover the configuration enumerated in Table 1. ADSSEC created an initial population of 20 random individuals. The master process used 20 slaves processes for evaluating offspring, either synchronously producing offspring at each generation or asynchronously creating new offspring as each node became available.

ADSSEC selected parents uniformly for either recombination or mutation – with a mutation probability of 0.10 and, subsequently, a recombination rate of 0.90 – and used a shared crowding method for survival selection. ADSSEC was configured to terminate after 1000 evaluations and restart after 100 evaluations without improvement.

We executed ADSSEC on a machine with dual Intel Xeon E5-2630 v3 2.4 GHz octa-core processors and 128 GB 2133 MHz DDR4 RDIMM ECC RAM running Ubuntu 14.04. Both the synchronous and asynchronous models were run 8 times on Datasets A, B, C, and D.

5. RESULTS AND DISCUSSION

The total user time across all processes was measured from each run of ADSSEC for both the synchronous model and asynchronous model on the datasets. The variance in the number of decisions needed to solve the instances in each dataset directly influences the variation in evaluation time for each heuristic. Increased variation in evaluation times allows for greater speed-ups in the asynchronous model over the synchronous approach. As illustrated in Figure 1, the growth in variance of MiniSat decisions in each dataset results in more evident speed-ups for the asynchronous evolution times. Additionally, there may be a proportional increase in variation of evolution time for both models.

The results from ANOVA tests confirm the improvement provided by the asynchronous method (see Table 2); as the p-values are all approximately zero, there is very high confidence in this conclusion. In Dataset D, the synchronous method needed an average of 5.24 hours to complete the

same number of evaluations that asynchronous finished in an average of 3.19 hours. These results were obtained where the longest time to solve an instance is approximately five seconds. In the real-world, industrial instances can require several minutes to hours to complete. Employing the asynchronous evolution when training ADSSEC on datasets containing those instances would measure speed-ups in CPU days or weeks.

Table 2: ANOVA results of evolution time in seconds comparing both models of ADSSEC. (The variance is measured in seconds^2.)

	Synchronous	**Asynchronous**
Dataset A		
Mean	515.1038	435.1900
Variance	35.4130	33.4400
P-value	0.0000	
Dataset B		
Mean	1,647.6300	1,198.9300
Variance	11,554.0380	1,455.4835
P-value	0.0000	
Dataset C		
Mean	15,453.0900	9,260.6925
Variance	493,279.9961	112,446.0049
P-value	0.0000	
Dataset D		
Mean	18,865.0775	11,470.6363
Variance	3,539,070.1957	230,474.1322
P-value	0.0000	

6. CONCLUSION

For EAs where the fitness evaluation times can vary drastically, especially in the case of hyper-heuristics, just parallelizing the evaluations to minimize evolution time is not always sufficient. The asynchronous approach to modelling hyper-heuristics will certainly provide a significant speed-up in evolution time over the synchronous alternative. As the variation in evaluation time increases, so does the speed-up. Instances from the SAT competitions and within industrial problem classes often have large variations in solving times. The asynchronous approach is necessary for addressing training systems similar to ADSSEC on the industrial instances. This paper has provided empirical evidence of the substantial performance gains of the asynchronous approach demonstrating that APEAs are the future of hyper-heuristics.

7. FUTURE WORK

- Measuring the rate of convergence between the models may conclude whether asynchronous or synchronous approaches produce superior heuristics in less time. While asynchronous evolution completed the same number of evaluations in less time, more evidence is needed to conclude that the quality of heuristics produced will match the synchronous counterpart at any set number of evaluations.
- Use either different datasets or larger sample sizes to determine how the generated heuristic quality varies with the training instances.
- While hyper-heuristics can be computationally expensive, providing some level of automated parameter tuning may further reduce the evolution time. This is dependent on the sensitivity of the parameters or inputs selected for automation.
- Comparing ADSSEC to parameter optimization tools, such as those mentioned in the related work section, applied to CDCL SAT solvers may determine the benefits and limitations of this asynchronous approach for different datasets. Additionally, incorporating parameter optimization as a post-processing step or even throughout evolution may develop more targeted CDCL SAT solvers.

8. REFERENCES

[1] E. Alba. Parallel evolutionary algorithms can achieve super-linear performance. *Information Processing Letters*, 82(1):7–13, 2002.

[2] E. Alba and M. Tomassini. Parallelism and Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5):443–462, 2002.

[3] E. Alba and J. M. Troya. Analyzing Synchronous and Asynchronous Parallel Distributed Genetic Algorithms. *Future Generation Computer Systems*, 17(4):451–465, 2001.

[4] M. Bader-El-Den and R. Poli. Generating SAT Local-Search Heuristics Using a GP Hyper-Heuristic Framework. In *Artificial Evolution*, volume 4926 of *Lecture Notes in Computer Science*, pages 37–49, Tours, France, Oct. 2008. Springer Berlin Heidelberg.

[5] A. Biere and A. Fröhlich. Evaluating CDCL Restart Schemes. In *Proceedings of the International Workshop on Pragmatics of SAT (POS'15)*, Austin, TX, Sept. 2015. 16 pages.

[6] A. Biere and A. Fröhlich. Evaluating CDCL Variable Scoring Schemes. In *Theory and Applications of Satisfiability Testing–SAT 2015*, volume 9340 of *Lecture Notes in Computer Science*, pages 405–422. Springer International Publishing, Austin, TX, USA, Sept. 2015.

[7] S. Bleuler, M. Brack, L. Thiele, and E. Zitzler. Multiobjective Genetic Programming: Reducing Bloat Using SPEA2. In *Proceedings of the 2001 Congress on Evolutionary Computation*, volume 1, pages 536–543. IEEE, 2001.

[8] A. W. Churchill, P. Husbands, and A. Philippides. Tool Sequence Optimization using Synchronous and Asynchronous Parallel Multi-Objective Evolutionary Algorithms with Heterogeneous Evaluations. In *2013 IEEE Congress on Evolutionary Computation (CEC)*, pages 2924–2931. IEEE, 2013.

[9] J. J. Durillo, A. J. Nebro, F. Luna, and E. Alba. A Study of Master-Slave Approaches to Parallelize NSGA-II. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008.

[10] N. Eén and N. Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing–SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518, Santa Margherita Ligure, Italy, May 2003. Springer Berlin Heidelberg.

[11] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, second edition, 2015. page 93.

[12] S. Falkner, M. Lindauer, and F. Hutter. SpySMAC: Automated Configuration and Performance Analysis of SAT Solvers. In *Theory and Applications of Satisfiability Testing–SAT 2015*, volume 9340 of *Lecture Notes in Computer Science*, pages 215–222. Springer International Publishing, Austin, TX, USA, Sept. 2015.

[13] A. S. Fukunaga. Evolving Local Search Heuristics for SAT Using Genetic Programming. In *Genetic and Evolutionary Computation Conference–GECCO 2004*, volume 3103 of *Lecture Notes in Computer Science*, pages 483–494, Seattle, WA, USA, June 2004. Springer Berlin Heidelberg.

[14] A. S. Fukunaga. Automated Discovery of Local Search Heuristics for Satisfiability Testing. *Evolutionary Computation*, 16(1):31–61, Apr. 2008.

[15] A. S. Fukunaga. Massively Parallel Evolution of SAT Heuristics. In *2009 IEEE Congress on Evolutionary Computation (CEC)*, pages 1478–1485, Trondheim, Norway, May 2009. IEEE.

[16] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization*, volume 6683 of *Lecture Notes in Computer Science*, pages 507–523. Springer Berlin Heidelberg, Rome, Italy, Jan. 2011.

[17] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, Sept. 2009.

[18] R. H. Kibria and Y. Li. Optimizing the Initialization of Dynamic Decision Heuristics in DPLL SAT Solvers Using Genetic Programming. In *Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 331–340. Springer Berlin Heidelberg, Budapest, Hungary, Apr. 2006.

[19] M. A. Martin, A. R. Bertels, and D. R. Tauritz. Asynchronous Parallel Evolutionary Algorithms: Leveraging Heterogeneous Fitness Evaluation Times for Scalability and Elitist Parsimony Pressure. In *Proceedings of the 17th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '15)*, pages 1429–1430, Madrid, Spain, July 2015. ACM.

[20] M. A. Martin and D. R. Tauritz. A Problem Configuration Study of the Robustness of a Black-Box Search Algorithm Hyper-Heuristic. In *Proceedings of the 16th Annual Conference on Genetic and Evolutionary Computation (GECCO '14)*, pages 1389–1396. ACM, 2014.

[21] M. A. Martin and D. R. Tauritz. Multi-Sample Evolution of Robust Black-Box Search Algorithms. In *Proceedings of the 16th Annual Conference on Genetic and Evolutionary Computation (GECCO '14)*, pages 195–196. ACM, 2014.

[22] M. A. Martin and D. R. Tauritz. Hyper-Heuristics: A Study On Increasing Primitive-Space. In *Proceedings of the 17th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '15)*, pages 1051–1058, Madrid, Spain, July 2015. ACM.

[23] J. Miller. What bloat? Cartesian Genetic Programming on Boolean problems. In *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 295–302, 2001.

[24] M. Oussaidene, B. Chopard, O. V. Pictet, and M. Tomassini. Parallel Genetic Programming and its application to trading model induction. *Parallel Computing*, 23(8):1183–1198, 1997.

[25] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class. In *Genetic Programming*, volume 8599 of *Lecture Notes in Computer Science*, pages 137–149. Springer Berlin Heidelberg, Granada, Spain, Apr. 2014.

[26] J. Petke, W. B. Langdon, and M. Harman. Applying Genetic Improvement to MiniSAT. In *Search Based Software Engineering*, volume 8084 of *Lecture Notes in Computer Science*, pages 257–262. Springer Berlin Heidelberg, St. Petersburg, Russia, Aug. 2013.

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

[28] E. O. Scott and K. A. De Jong. Evaluation-Time Bias in Asynchronous Evolutionary Algorithms. In *Proceedings of the 17th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '15)*, pages 1209–1212, Madrid, Spain, July 2015. ACM.

[29] M. Yagoubi and M. Schoenauer. Asynchronous Master/Slave MOEAs and Heterogeneous Evaluation Costs. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation Conference (GECCO '12)*, pages 1007–1014. ACM, 2012.

[30] M. Yagoubi, L. Thobois, and M. Schoenauer. Asynchronous Evolutionary Multi-Objective Algorithms with Heterogeneous Evaluation Costs. In *2011 IEEE Congress on Evolutionary Computation (CEC)*, pages 21–28. IEEE, 2011.