An Efficient Evolutionary Algorithm for Solving Incrementally Structured Problems

Jason Ansel Maciej Pacula Saman Amarasinghe Una-May O'Reilly
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA, USA
{jansel,mpacula,saman,unamay}@csail.mit.edu

ABSTRACT

Many real world problems have a structure where small problem instances are embedded within large problem instances, or where solution quality for large problem instances is loosely correlated to that of small problem instances. This structure can be exploited because smaller problem instances typically have smaller search spaces and are cheaper to evaluate. We present an evolutionary algorithm, INCREA, which is designed to incrementally solve a large, noisy, computationally expensive problem by deriving its initial population through recursively running itself on problem instances of smaller sizes. The INCREA algorithm also expands and shrinks its population each generation and cuts off work that doesn't appear to promise a fruitful result. For further efficiency, it addresses noisy solution quality efficiently by focusing on resolving it for small, potentially reusable solutions which have a much lower cost of evaluation. We compare INCREA to a general purpose evolutionary algorithm and find that in most cases INCREA arrives at the same solution in significantly less time.

Categories and Subject Descriptors

I.2.5 [**Artificial Intelligence**]: Programming Languages and Software; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms

Algorithms, Experimentation, Languages

1. INTRODUCTION

An off-the-shelf evolutionary algorithm (EA) does not typically take advantage of shortcuts based on problem properties and this can sometimes make it impractical because it takes too long to run. A general shortcut is to solve a small instance of the problem first then reuse the solution in a compositional manner to solve the large instance which is of interest. Usually solving a small instance is both simpler (because the search space is smaller) and less expensive (because the evaluation cost is lower). Reusing a sub-solution or using it as a starting point makes finding a solution to a larger instance quicker. This shortcut is particularly advantageous if solution evaluation cost grows with instance size. It becomes more advantageous if the evaluation result is noisy or highly variable which requires additional evaluation sampling.

This shortcut is vulnerable to local optima: a small instance solution might become entrenched in the larger solution but not be part of the global optimum. Or, non-linear effects between variables of the smaller and larger instances may imply the small instance solution is not reusable. However, because EAs are stochastic and population-based they are able to avoid potential local optima arising from small instance solutions and address the potential non-linearity introduced by the newly active variables in the genome.

In this contribution, we describe an EA called INCREA which incorporates into its search strategy the aforementioned shortcut through incremental solving. It solves increasingly larger problem instances by first activating only the variables in its genome relevant to the smallest instance, then extending the active portion of its genome and problem size whenever an instance is solved. It shrinks and grows its population size adaptively to populate a gene pool that focuses on high performing solutions in order to avoid risky, excessively expensive, exploration. It assumes that fitness evaluation is noisy and addresses the noise adaptively throughout the tuning process.

Problems of practical value with this combination of opportunity and requirements exist. We will exemplify the INCREA technique by solving a software engineering problem known as autotuning. We are developing a programming language named Petabricks which supports multi-core programming. In our case, autotuning arises as a final task of program compilation and occurs at program installation time. Its goal is to select parameters and algorithmic choices for the program to make it run as fast as possible. Because a program can have varying size inputs, INCREA tunes the program for small input sizes before incrementing them up to the point of the maximum expected input sizes.

We proceed in the following manner: Section 2 describes compilation autotuning, and elaborates upon how it exhibits the aforementioned properties of interest. Section 3 describes the design of INCREA. Section 4 discuss its relation to other relevant EAs. Section 5 experimentally compares INCREA to a general purpose EA. Section 6 concludes.

2. AUTOTUNING

When writing programs that require performance, the programmer is often faced with many alternate ways to implement a set of algorithms. Most often the programmer chooses a single set based on manual testing. Unfortunately, with the increasing diversity in architectures, finding one set that performs well everywhere is impossible for many problems. A single program might be expected to run on a small, embedded cell phone, a multi-core desktop or server, a cluster, a grid, or the cloud. The appropriate algorithmic choices may be very different for each of these architectures.

An example of this type of hand coding of algorithms can be found in the std::sort routine found in the C++ Standard Template Library. This routine performs merge sort until there are less than 15 elements in a recursive call then switches to insertion sort. This cutoff is hardcoded, even though values 10 times larger can perform better on modern architectures and entirely different algorithms, such as bitonic sort, can perform better on parallel vector machines [3].

This motivates the need for an autotuning capability. The programmer should be able to provide a choice of algorithms, describe a range of expected data sizes, and then pass to the compiler the job of experimentally determining, for any specific architecture, what cutoff points apply and what algorithms are appropriate for different data ranges.

The autotuners in this paper operate on programs written in the PetaBricks programming language [3] which allows the programmer to express algorithmic choices at the language level without committing to any single one, or any combination thereof. The task of choosing the fastest algorithm set is delegated to an auotuner which makes the decision automatically depending on input sizes to each algorithm (and an overall target input size) and a target architecture.

2.1 The Autotuning Problem

The autotuner must identify selectors that will determine which choice of an algorithm will be used during a program execution so that the program executes as fast as possible. Formally, a selector s consists of $\vec{C_s} = [c_{s,1}, \ldots, c_{s,m-1}] \cup \vec{A_s} = [\alpha_{s,1}, \ldots, \alpha_{s,m}]$ where $\vec{C_s}$ are the ordered interval boundaries (cutoffs) associated with algorithms $\vec{A_s}$. During program execution the runtime function SELECT chooses an algorithm depending on the current input size by referencing the selector as follows:

$SELECT(input, s) = \alpha_{s,i}$ s.t. $c_{s,i} > size(input) \geq c_{s,i-1}$ where
$c_{s,0} = min(size(input))$ and $c_{s,m} = max(size(input))$.

The components of $\vec{A_s}$ are indices into a discrete set of applicable algorithms available to s, which we denote $Algorithms_s$. The maximum number of intervals is fixed by the PetaBricks compiler. An example of a selector for a sample sorting algorithm is shown in Figure 1.

In addition to algorithmic choices, the autotuner tunes parameters such as blocking sizes, sequential/parallel cutoffs and the number of worker threads. Each tunable is either a discrete value of a small set indexed by an integer or a integer in some positive bounded range.

Formally, given a program P, hardware H and input size n, the autotuner must identify the vector of selectors and

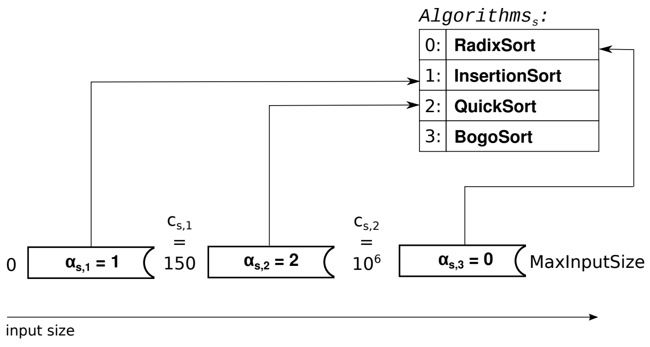


Figure 1: A selector for a sample sorting algorithm where $\vec{C_s} = [150, 10^6]$ and $\vec{A_s} = [1, 2, 0]$. The selector selects the $InsertionSort$ algorithm for input sizes in the range $[0; 150)$, $QuickSort$ for input sizes in the range $[150, 10^6)$ and $RadixSort$ for $[10^6, MAXINT)$. $BogoSort$ was suboptimal for all input ranges and is not used.

vector of tunables such that the following objective function $executionTime$ is satisfied:

$$\arg\min_{\bar{s},\bar{t}} executionTime(P, H, n)$$

2.2 Properties of the Autotuning Problem

Three properties of autotuning influence the design of an autotuner. First, the cost of fitness evaluation depends heavily on on the input data size used when testing the candidate solution. The autotuner does not necessarily have to use the target input size. For efficiency it could use smaller sizes to help it find a solution to the target size because is generally true that smaller input sizes are cheaper to test on than larger sizes, though exactly how much cheaper depends on the algorithm. For example, when tuning matrix multiply one would expect testing on a 1024×1024 matrix to be about 8 times more expensive than a 512×512 matrix because the underlying algorithm has $O(n^3)$ performance. While solutions on input sizes smaller than the target size sometimes are different from what they would be when they are evolved on the target input size, it can generally be expected that relative rankings are robust to relatively small changes in input size. This naturally points to "bottom-up" tuning methods that incrementally reuse smaller input size tests or seed them into the initial population for larger input sizes.

Second, in autotuning the fitness of a solution is its fitness evaluation cost. Therefore the cost of fitness evaluation is dependant on the quality of a candidate algorithm. A highly tuned and optimized program will run more quickly than a randomly generated one and it will thus be fitter. This implies that fitness evaluations become cheaper as the overall fitness of the population improves.

Third, significant to autotuning well is recognizing the fact that fitness evaluation is noisy due to details of the parallel micro-architecture being run on and artifacts of concurrent activity in the operating system. The noise can come from many sources, including: caches and branch prediction; races between dependant threads to complete work; operating system artifacts such as scheduling, paging, and I/O;

and, finally, other competing load on the system. This leads to a design conflict: an autotuner can run fewer tests, risking incorrectly evaluating relative performance but finishing quickly, or it can run many tests, likely be more accurate but finish too slowly. An appropriate strategy is to run more tests on less expensive (i.e. smaller) input sizes.

The INCREA exploits incremental structure and handles the noise exemplified in autotuning. We now proceed to describe a INCREA for autotuning.

3. A BOTTOM UP EA FOR AUTOTUNING

Representation

The INCREA genome, see Figure 2, encodes a list of selectors and tunables as integers each in the range $[0, MaxVal)$ where $MaxVal$ is the cardinality of each algorithm choice set for algorithms and $MaxInputSize$ for cutoffs. Each tunable has a $MaxVal$ which is the cardinality of its value set or a bounded integer depending on what it represents.

In order to tune programs of different input sizes the genome represents a solution for maximum input size and throughout the run increases the "active" portion of it starting from the selectors and tunables relevant to the smallest input size. It has length $(2m + 1)k + n$, where k is the number of selectors, m the number of interval cutoffs within each selector and n the number of other tunables defined for the PetaBricks program. As the algorithm progresses the number of "active" cutoff and algorithm pairs, which we call "choices" for each selector in the genome starts at 1 and then is incremented in step with the algorithm doubling the current input size each generation.

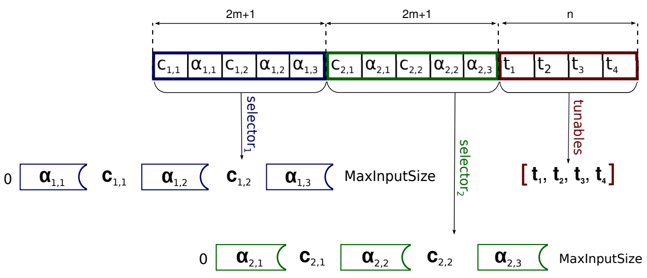

2m+1
2m+1
n
$c_{1,1}$ $\alpha_{1,1}$ $c_{1,2}$ $\alpha_{1,2}$ $\alpha_{1,3}$ $c_{2,1}$ $\alpha_{2,1}$ $c_{2,2}$ $\alpha_{2,2}$ $\alpha_{2,3}$ t_1 t_2 t_3 t_4
selector$_1$
selector$_2$
tunables
0 $\alpha_{1,1}$ $c_{1,1}$ $\alpha_{1,2}$ $c_{1,2}$ $\alpha_{1,3}$ MaxInputSize
$[t_1, t_2, t_3, t_4]$
0 $\alpha_{2,1}$ $c_{2,1}$ $\alpha_{2,2}$ $c_{2,2}$ $\alpha_{2,3}$ MaxInputSize

Figure 2: A sample genome for $m = 2$, $k = 2$ and $n = 4$. Each gene stores either a cutoff $c_{s,i}$, an algorithm $\alpha_{s,i}$ or a tunable value t_i.

Fitness evaluation

The fitness of a genome is the inverse of the corresponding program's execution time. The execution time is obtained by timing the PetaBricks program for a specified input size.

Top level Strategy

Figure 3 shows top level pseudocode for INCREA. The algorithm starts with a "parent" population and an input size of 1 for testing each candidate solution. All choices and tunables are initially set to algorithm 0 and cutoff of MAX_INT. The choice set is grown through mutation on a per candidate basis. The input size used for fitness evaluation doubles each generation.

A generation consists of 2 phases: exploration, and downsizing. During exploration, a random parent is used to generate a child via mutation. Only active choices and tunables are mutated in this process. The choice set may be enlarged. The child is added to the population only if it is determined to be fitter than its parent. The function "fitter" which tests for this condition increases trials of the parent or child to improve confidence in their relative fitnesses. Exploration repeatedly generates a child and tests it against its parent for some fixed number of MutationAttempts or until the population growth reaches some hard limit.

During downsizing, the population, which has potentially grown during exploration, is pruned down to its original size once it is ranked. The "rankThenPrune" function efficiently performs additional fitness tests only as necessary to determine a ranking of which it is reasonably certain.

This strategy is reminiscent but somewhat different from a $(\mu + \lambda)$ES [5]. The $(\mu + \lambda)$ES creates a pool of λ offspring each generation by random draws from the parent population of size μ. Then both offspring and parents are combined and ranked for selection into the next generation. The subtle differences in INCREA are that 1) in a "steady state" manner, INCREA inserts any child which is better than its parent immediately into the population while parents are still being drawn, and 2) a child must be fitter than its parent before it gains entry into the population. The subsequent ranking and pruning of the population matches the selection strategy of $(\mu + \lambda)$ES.

Doubling the input size used for fitness evaluation at each generation allows the algorithm to learn good selectors for smaller ranges before it has to find ones for bigger ranges. It supports subsolution reuse and going forward from a potentially good, non-random starting point. Applying mutation to only the active choice set and tunables while input size is doubling brings additional efficiency because this narrows down the search space while concurrently saving on the cost of fitness evaluations because testing solutions on smaller inputs sizes is cheaper.

```
populationSize = popLowSize
inputSizes = [1, 2, 4, 8, 16, ...., maxInputSize]
initialize population(maxGenomeLength)
for gen = 1 to log(maxInputSize)
  /* exploration phase: population and active
     choices may increase */
  inputSize = inputSizes[gen]
  for j = 1 to mutationAttempts
    parent = random draw from population
    activeChoices = getActiveChoices(parent)
    /* active choices could grow */
    child = mutate(parent, activeChoices)
    /* requires fitness evaluations */
    if fitter(child, parent, inputSize)
      population = add(population, child)
      if length(population) >= popHighSize
        exit exploration phase
  end /* exploration phase */
  /* more testing */
  population = rankThenPrune(population,
                             popLowSize,
                             inputSize)
  /* discard all past fitness evaluations */
  clearResults(population)
end /* generation loop*/
return fittest population member
```

Figure 3: Top level strategy of INCREA.

Mutation Operators

The mutators perform different operations based on the type of value being mutated. For an algorithmic choice, the new value is drawn from a uniform probability distribution $[0, ||Algorithms_s|| - 1]$. For a cutoff, the existing value is scaled by a random value drawn from a log-normal distribution, i.e. doubling and halving the existing value are equally likely. The intuition for a log-normal distribution is that small changes have larger effects on small values than large values in autotuning. We have confirmed this intuition experimentally by observing much faster convergence times with this type of scaling.

The INCREA mutation operator is only applied to choices that are in the active choice list for the genome. INCREA has one specialized mutation operator that adds another choice to the active choice list of the genome and sets the cutoff to 0.75 times the current input size while choosing the algorithm randomly. This leaves the behavior for smaller inputs the same, while changing the behavior for the current set of inputs being tested. It also does not allow a new algorithm to be the same as the one for the next lower cutoff.

Noisy Fitness Strategies

Because INCREA must also contend with noisy feedback on program execution times, it is bolstered to evaluate candidate solutions multiple times when it is ranking any pair. Because care must be taken not to test too frequently, especially if the input data size is large, it uses an adaptive sampling strategy [1, 8, 9, 16] . The boolean function "fitter", see Figure 4, takes care of this concern by running more fitness trials for candidates $s1$ and $s2$ under two criteria. The first criterion is a t-test [13]. When the t-test result has a confidence, i.e. p-value less than 0.05, $s1$ and $s2$ are considered different and trials are halted. If the t-test cannot confirm difference, least squares is used to fit a normal distribution to the percentage difference in the mean execution time of the two algorithms. If this distribution estimates there is a 95% probability of less than a 1% difference, the two candidates' fitnesses are considered to be the same. There is also a parameterized hard upper limit on trials.

The parent ranking before pruning, in function "rankThenPrune", is optimized to minimize the number of additional fitness evaluations. First, it ranks the entire population by mean performance without running any additional trials. It then splits the ranking at the *populationLowSize* element into a $KEEP$ list and a $DISCARD$ list. Next, it sorts the $KEEP$ list by calling the "fitter" function (which may execute more fitness trials). Next, it compares each candidate in the $DISCARD$ list to the *populationLowSize* element in the $KEEP$ list by calling the "fitter" function. If any of these candidates are faster, they are moved to the $KEEP$ list. Finally, the $KEEP$ list is sorted again by calling "fitter" and the first *populationLowSize* candidates are the result of the pruning.

This strategy avoids completely testing the elements of the population that will be discarded. It allocates more testing time to the candidate that will be kept in the population. It also exploits the fact that comparing algorithms with larger differences in performance is cheaper than comparing algorithms with similar performance.

4. RELATED WORK

There exists a large variety of work related to PetaBrick's approach of autotuning computer programs. PHiPAC [6] is an autotuning system for dense matrix multiply, generating portable C code and search scripts to tune for specific systems. ATLAS [18] utilizes empirical autotuning to produce a cache-contained matrix multiply, which is then used in larger matrix computations in BLAS and LAPACK. FFTW [10] uses empirical autotuning to combine solvers for FFTs. Other autotuning systems include SPARSITY [12] for sparse matrix computations, SPIRAL [14] for digital signal processing, UHFFT [2] for FFT on multicore systems, and OSKI [17] for sparse matrix kernels.

Layered learning, [15], used for robot soccer, is broadly related to our work. Like INCREA, layered learning is used when a mapping directly from inputs to outputs is not tractable and when a task can be decomposed to be solved bottom up. In layered learning however, composition occurs through learning, in the general sense of abstracting and solving the local concept-learning task. (See [11] where genetic programming is used for learning.) INCREA combines optimizations, in contrast. Both approaches use domain specific knowledge to determine appropriate learning granularity: input size doubling in INCREA) and subtask definition in layered learning. Layered learning occurs separately on each hand designed level with a hand designed interface between each level. In contrast, INCREA incorporates the entire composition into one algorithm which automatically lengthens the genome only as needed.

Using an adaptive sampling strategy for fitness estimation dates back to [1]. A combination of approaches from [9, 16] inform INCREA's strategy. In [9] a t-test is evaluated and found to be effective when an appropriate population size is not known. In [16] individuals are further evaluated only if there is some chance that the outcome of the tournaments they participate in can change. The GPEA may derive some of its robustness to noise from its use of a relatively large population. See [7, 4] for discussions of this robustness.

5. EXPERIMENTAL EVALUATION

We now compare INCREA to a general purpose EA we call GPEA on 4 Petabricks benchmarks: `sort` (for 2 target input sizes), `matmult` which is dense matrix multiply, and `eig` which solves for symmetric eigenvalues.

5.1 GPEA

The GPEA uses the same genome representation and operators of INCREA. All selector choices are always active. It initializes all population members with values drawn from the distributions used by the mutation operator. It then loops evaluating the fitness of each member once, performing tournament selection and applying crossover with $p_{xo} = 1.0$ then mutation with probability p_{μ}. Crossover swaps algorithms while cutoffs and tunables are swapped or changed to a random value in between those of the two parents' genes. Extrapolating from [7, 4], GPEA's significant population size (100 in our experiments) should provide some robustness to fitness case noise.

5.2 Experimental Setup

We performed all tests on multiple identical 8-core, dual-Xeon X5460, systems clocked at 3.16 GHz with 8 GB of RAM. The systems were running Debian GNU/Linux 5.0.3

```
function fitter(s1, s2, inputSize)
  while s1.evalCount < evalsLowerLimit
    evaluateFitness(s1, inputSize)
  end
  while s2.evalCount < evalsLowerLimit
    evaluateFitness(s2, inputSize)
  end
  while true
    /* Single tailed T-test assumes each sample's mean is normally distributed.
       It reports probability that sample means are same under this assumption  */
    if ttest(s1.evalsResults, s2.evalResults) < PvalueLimit /* statistically different */
      return mean(s1.evalResults) > mean(s2.evalResults)
    end
    /* Test2Equality: Use least squares to fit a normal distribution to the percentage
       difference in the mean performance of the two algorithms. If this
       distribution estimates there is a 95% probability of less than a 1%
       difference in true means, consider the two algorithms the same. */
    if Test2Equality(s1.evalResults, s2.evalResults)
      return false
    end
    /* need more information, choose s1 or s2 based on the highest expected
       reduction in standard error */
    whoToTest = mostInformative(s1, s2);
    if whoToTest == s1 and s1.testCount < evalsUpperLimit
      evaluateFitness(s1, inputSize)
    elif s2.testCount < evalsUpperLimit
      evaluateFitness(s2, inputSize)
    else
      /* inconclusive result, no more evals left */
      return false
    end
  end /* while */
end /* fitter */
```

Figure 4: Pseudocode of function "fitter".

Parameter	Value
confidence required	70%
max trials	5
min trials	1
population high size	10
population low size	2
mutationAttempts	6
standard deviation prior	15%

(a) INCREA

Parameter	Value
mutation rate	0.5
crossover rate	1.0
population size	100
tournament size	10
generations	100
evaluations per candidate	1

(b) GPEA

Figure 5: INCREA and GPEA Parameter Settings.

with kernel version 2.6.26. For each test, we chose a target input size large enough to allow parallelism, and small enough to converge on a solution within a reasonable amount of time. Parameters such as the mutation rate, population size and the number of generations were determined experimentally and kept constant between benchmarks. Parameter values we used are listed in Figure 5.

5.3 INCREA vs GPEA

In practice we might choose parameters of either INCREA or GPEA to robustly ensure good autotuning or allow the programmer to vary them while tuning a particular problem and architecture. In the latter case, considering how quickly the tuner converges to the final solution is important. To more extensively compare the two tuners, we ran each tuner 30 times for each benchmark.

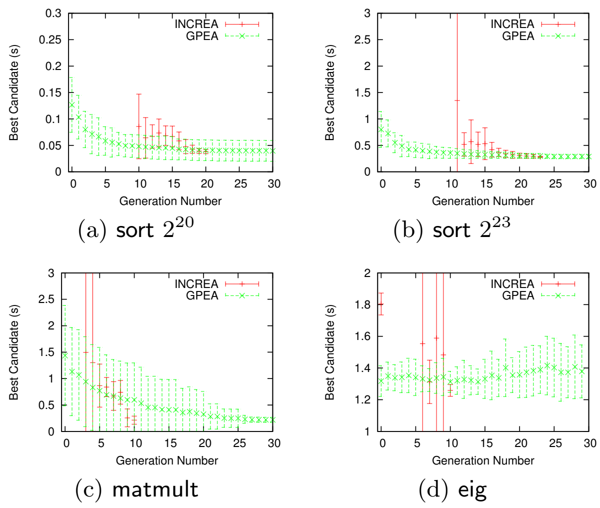
Table 1 compares the tuners mean performance with 30 runs based on time to convergence and the performance of the final solution. To account for noise, time to convergence is calculated as the first time that a candidate was found that was within 5% of the best fitness achieved. For all of the benchmarks except for eig, both tuners arrive at nearly the same solutions, while for eig INCREA finds a slightly better solution. For eig and matmult, INCREA converges an order of magnitude faster than GPEA. For sort, GPEAconverges faster on the small input size while INCREA converges faster on the larger input size. If one extrapolates convergences times to larger input sizes, it is clear that INCREA scales a lot better than GPEA for sort.

		INCREA	GPEA	SS?
sort-2^{20}	Convergence	1464.7 ± 1992.0	599.2 ± 362.9	YES ($p = 0.03$)
	Performance	0.037 ± 0.004	0.034 ± 0.014	NO
sort-2^{23}	Convergence	2058.2 ± 2850.9	2480.5 ± 1194.5	NO
	Performance	0.275 ± 0.010	0.276 ± 0.041	NO
matmult	Convergence	278.5 ± 185.8	2394.2 ± 1931.0	YES ($p = 10^{-16}$)
	Performance	0.204 ± 0.001	0.203 ± 0.001	NO
eig	Convergence	92.1 ± 66.4	627.4 ± 530.2	YES ($p = 10^{-15}$)
	Performance	1.240 ± 0.025	1.250 ± 0.014	YES ($p = 0.05$)

Table 1: Comparison of INCREA and GPEA in terms of mean time to convergence in seconds and in terms of execution time of the final configuration. Standard deviation is shown after the ± symbol. The final column is statistical significance determined by a t-test. (Lower is better)

Figure 6 shows aggregate results from 30 runs for both INCREA and GPEA on each benchmark. INCREA generally has a large amount of variance in its early generations, because those generations are based on smaller input sizes that may have different optimal solutions than the largest input size. However, once INCREA reaches its final generation it exhibits lower variance than than GPEA. GPEA tends to converge slowly with gradually decreasing variance. Note that the first few generations for INCREA are not shown because, since it was training on extremely small input sizes, it finds candidates candidates that exceed the timeout set by our testing framework when run on the largest input size. These early generations account for a only a small amount of the total training time.

In 6(a) the INCREA's best candidate's execution time displays a "hump" that is caused because it finds optima for smaller input sizes that are not reused in the optimal solution for the target input size.



(a) sort 2^{20} (b) sort 2^{23} (c) matmult (d) eig

Figure 6: Execution time for target input size with best individual of generation. Mean and standard deviation (shown in error bars) with 30 runs.

Using sort-2^{20}, in Figure 7(a) we examine how many tests are halted by each tuner, indicating very poor solutions. The timeout limit for both algorithms is set to be the same factor of the time of the current best solution. However, in GPEA this will always be a test with the target input size whereas with INCREA it is the current input size (which is at least half the time, half as large). Almost half of GPEA's initial population were stopped for timing out, while INCREA experiences most of its timeouts in the later generations where the difference between good and bad solutions grows with the larger input sizes. We also examine in Figure 7(b) how much the population grew each generation during the exploration phase. For INCREA the population expansion during exploration is larger in the middle generations as it converges to a final solution.

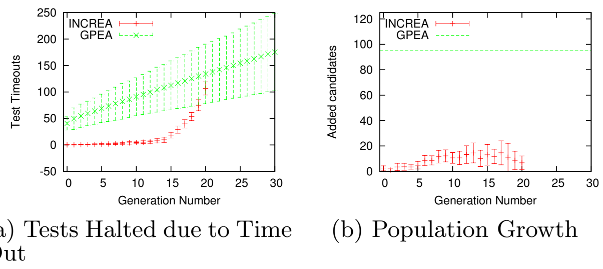
5.4 Representative runs

We now select a representative run for each benchmark to focus on run dynamics.

sort*: Sorting*

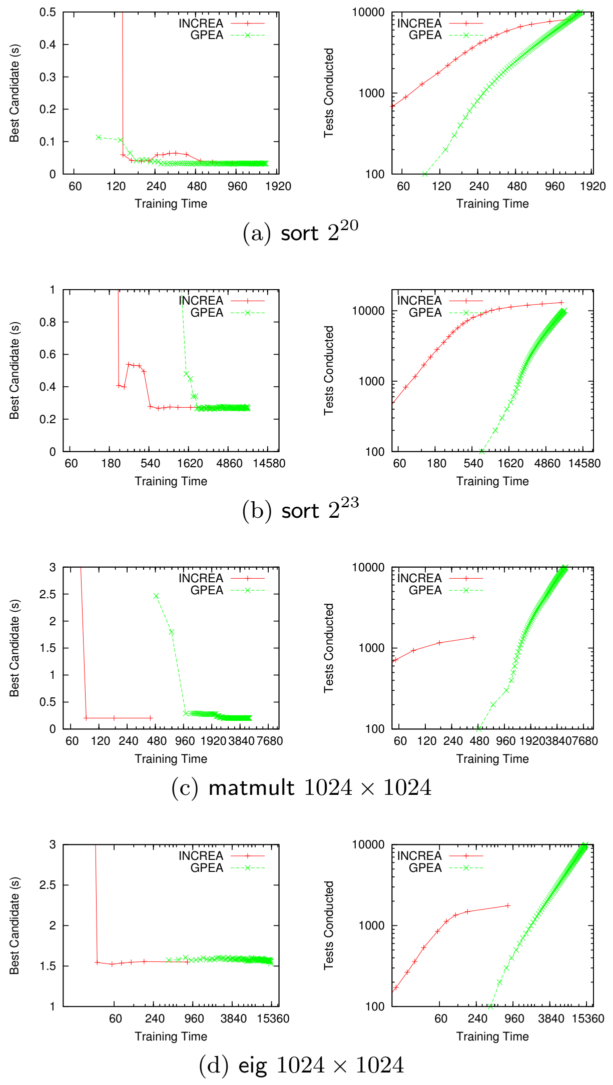
Figures 8(a) and 8(b) show results from a representative run of each autotuner with two different target input sizes respectively. The benchmark consists of insertion-sort, quick-sort, radix sort, and 2/4/8/16/32-way merge-sorts. On this Xeon system, sort is relatively easy to tune because the optimal solution is relatively simple and the relative costs of the different algorithms are similar.

For the 2^{20} benchmark, both INCREA and GPEA consistently converge to a very similar solution which consists of small variations of quick-sort switching to insertion-sort at



(a) Tests Halted due to Time Out (b) Population Growth

Figure 7: Time out and population growth statistics of INCREA for 30 runs of sort on target input size 2^{20}. Error bars are mean plus and minus one standard deviation.



(a) sort 2^{20}

(b) sort 2^{23}

(c) matmult 1024×1024

(d) eig 1024×1024

Figure 8: Representative runs of INCREA and GPEA on each benchmark. The left graphs plot the execution time (on the target input size) of the best solution after each generation. The right graph plots the number of fitness evaluations conducted at the end of each generation. All graphs use seconds of training time as the x-axis.

somewhere between 256 and 512. Despite arriving at a similar place, the two tuners get there in a very different way. Table 2, lists the best algorithm for each tuner at each generation in the run first shown in Figure 8(a). INCREA starts with small input sizes, where insertion-sort alone performs well, and for generations 0 to 7 is generating algorithms that primarily use insertion-sort for the sizes being tested. From generations 8 to 16, it creates variants of radix-sort and quicksort that are sequential for the input sizes being tested. In generation 17 it switches to a parallel quick sort and proceeds to optimize the cutoff constants on that for the remaining rounds. The first two of these major phases are locally optimal for the smaller input sizes they are trained on.

GPEA starts with the best of a set of random solutions, which correctly chooses insertion-sort for small input sizes. It then finds, in generation 3, that quick-sort, rather than the initially chosen radix-sort, performs better on large input sizes within the tested range. In generation 6, it refines its solution by paralellizing quick-sort. The remainder of the training time is spent looking for the exact values of the algorithmic cutoffs, which converge to their final values in generation 29.

INCREA: sort		
Input size	Training Time (s)	Genome
2^0	6.9	Q 64 Q_p
2^1	14.6	Q 64 Q_p
2^2	26.6	I
2^3	37.6	I
2^4	50.3	I
2^5	64.1	I
2^6	86.5	I
2^7	115.7	I
2^8	138.6	I 270 R 1310 R_p
2^9	160.4	I 270 Q 1310 Q_p
2^{10}	190.1	I 270 Q 1310 Q_p
2^{11}	216.4	I 270 Q 3343 Q_p
2^{12}	250.0	I 189 R 13190 R_p
2^{13}	275.5	I 189 R 13190 R_p
2^{14}	307.6	I 189 R 17131 R_p
2^{15}	341.9	I 189 R 49718 R_p
2^{16}	409.3	I 189 R 124155 M^2
2^{17}	523.4	I 189 Q 5585 Q_p
2^{18}	642.9	I 189 Q 5585 Q_p
2^{19}	899.8	I 456 Q 5585 Q_p
2^{20}	1313.8	I 456 Q 5585 Q_p

GPEA: sort		
Gen	Training Time (s)	Genome
0	91.4	I 448 R
1	133.2	I 413 R
2	156.5	I 448 R
3	174.8	I 448 Q
4	192.0	I 448 Q
5	206.8	I 448 Q
6	222.9	I 448 Q 4096 Q_p
7	238.3	I 448 Q 4096 Q_p
8	253.0	I 448 Q 4096 Q_p
9	266.9	I 448 Q 4096 Q_p
10	281.1	I 371 Q 4096 Q_p
11	296.3	I 272 Q 4096 Q_p
12	310.8	I 272 Q 4096 Q_p
...		
27	530.2	I 272 Q 4096 Q_p
28	545.6	I 272 Q 4096 Q_p
29	559.5	I 370 Q 8192 Q_p
30	574.3	I 370 Q 8192 Q_p
...		

Table 2: Listing of the best genome of each generation for each autotuner for an example training run. The genomes are encoded as a list of algorithms (represented by letters), separated by the input sizes at which the resulting program will switch between them. The possible algorithms are: I = insertion-sort, Q = quick-sort, R = radix-sort, and M^x = x-way merge-sort. Algorithms may have a $_p$ subscript, which means they are run in parallel with a work stealing scheduler. For clarity, unreachable algorithms present in the genome are not shown.

We classified the possible mutation operations of INCREA and counted how frequently each was used in creating an offspring fitter than its parent. We identified specialized classes of operations that target specific elements of the genome. Table 3 lists statistics on each for the run first shown in Figure 8(a). The class most likely to generate an improved child scaled both algorithm and parallelism cutoffs. The

Mutation Class	Count	Times Tried	Effect on fitness		
			Positive	Negative	None
Make an algorithm active	8	586	2.7%	83.8%	13.5%
Lognormally scale a cutoff	11	1535	4.4%	50.4%	45.1%
Randomy switch an algorithm	12	1343	2.5%	50.4%	25.7%
Lognormally change a parallism cutoff	2	974	5.2%	38.7%	56.1%

Table 3: Effective and ineffective mutations when INCREA solves sort (target input size 2^{20}.)

class that changed just algorithms were less likely to cause improvement. Overall only 3.7% of mutations improved candidate fitness.

matmult*: Dense Matrix Multiply*

Figure 8(c) shows comparative results on matmult. The program choices are a naive matrix multiply and five different parallel recursive decompositions, including Strassen's Algorithm and a cache-oblivious decomposition. A tunable allows both autotuners to transpose combinations of inputs and outputs to the problem. To generate a valid solution, the autotuner must learn to put a base case in the lowest choice of the selector, otherwise it will create an infinite loop. Because many random mutations will create candidate algorithms that never terminate when tested, we impose a time limit on execution.

Both INCREA and GPEA converge to the same solution for matmult. This solution consists of transposing the second input and then doing a parallel cache-oblivious recursive decomposition down to 64×64 blocks which are processed sequentially.

While both tuners converge to same solution, INCREA arrives at it much more quickly. This is primarily due to the n^3 complexity of matrix multiply, which makes running small input size tests extremely cheap compared to larger input sizes and the large gap between fast and slow configurations. INCREA converges to the final solution in 88 seconds, using 935 trials, before GPEA has evaluated even 20% of its initial population of 100 trials. INCREA converges to a final solution in 9 generations when the input size has reached 256, while GPEA requires 45 generations at input size 1024. Overall, INCREA converges 32.8 times faster than GPEA for matrix multiply.

eig*: Symmetric Eigenproblem*

Figure 8(d) shows results for eig. Similar to matmult here INCREA performs much better because of the fast growth in cost of running tests. This benchmark is unique in that its timing results have higher variance due to locks and allocation in the underlying libraries used to implement certain mathematical functions. This high variance makes it difficult to autotune well, especially for GPEA which only runs a single test for each candidate algorithm. Both solutions found were of same structure, but INCREA was able to find better cutoffs values than the GPEA.

6. CONCLUSIONS

INCREA is an evolutionary algorithm that is efficiently designed for problems which are suited to incremental shortcuts and require them because of they have large search spaces and expensive solution evaluaton. It also efficiently handles problems which have noisy candidate solution quality. In the so called "real world", problems of this sort abound. A general purpose evolutionary algorithm ignores the incremental structure that exists in these problems and, while it may identify a solution, it wastes computation, takes too long and produces error prone results. INCREA solves smaller to larger problem instances as generations progress and it expands and shrinks its genome and population each generation. For further efficiency, it cuts off work that doesn't appear to promise a fruitful result. It addresses noisy solution quality efficiently by focusing on resolving it for small solutions which have a much lower cost of evaluation.

We have demonstrated and evaluated a INCREA by solving the autotuning problem of a research compiler called PetaBricks for multi-scale architectures. INCREA automatically determines user-defined runtime parameters and algorithmic choices that result in the fastest execution for the given hardware. We found that INCREA and a general purpose EA both achieve significant speedups on 3 benchmarks but INCREA is much more efficient because of its exploitation of the problem's bottom up structure and its greedy efficiencies. Our future work includes applying INCREA to problems in wind turbine farm layout.

7. ACKNOWLEDGMENTS

This work is partially supported by DOE Award DE-SC0005288. We would also like to thank the anonymous reviewers for their constructive feedback.

8. REFERENCES

[1] Akiko N. Aizawa and Benjamin W. Wah. Scheduling of genetic algorithms in a noisy environment. *Evolutionary Computation*, 2(2):97–122, 1994.

[2] Ayaz Ali, Lennart Johnsson, and Jaspal Subhlok. Scheduling FFT computation on SMP and multicore systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 293–301, New York, NY, USA, 2007. ACM.

[3] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009.

[4] Dirk V. Arnold and Hans-Georg Beyer. On the benefits of populations for noisy optimization. *Evolutionary Computation*, 11(2):111–127, 2003.

[5] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford University Press, New York NY, 1996.

[6] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM.

[7] Jurgen Branke. Creating robust solutions by means of evolutionary algorithms. In Agoston Eiben, Thomas Baeck, Marc Schoenauer, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature, PPSN V*, volume 1498 of *Lecture Notes in Computer Science*, pages 119–. Springer Berlin / Heidelberg, 1998.

[8] Jurgen Branke, Christian Schmidt, and Hartmut Schmec. Efficient fitness estimation in noisy environments. In *Proceedings of Genetic and Evolutionary Computation*, pages 243–250, 2001.

[9] Erick Cantu-Paz. Adaptive sampling for noisy problems. In *Genetic and Evolutionary Computation, GECCO 2004*, volume 3102 of *Lecture Notes in Computer Science*, pages 947–958. Springer Berlin / Heidelberg, 2004.

[10] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, February 2005. Invited paper, special issue on "Program Generation, Optimization, and Platform Adaptation".

[11] Steven M. Gustafson and William H. Hsu. Layered learning in genetic programming for a co-operative robot soccer problem. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 291–301, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[12] Eun-jin Im and Katherine Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Proceedings of the International Conference on Computational Science*, pages 127–136. Springer, 2001.

[13] Carol A. Markowski and Edward P. Markowski. Conditions for the effectiveness of a preliminary test of variance. 1990.

[14] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *IJHPCA*, 18(1):21–45, 2004.

[15] Peter Stone and Manuela Veloso. Layered learning. In Ramon Lopez de Montaras and Enric Plaza, editors, *Machine Learning: ECML 2000*, volume 1810 of *Lecture Notes in Computer Science*, pages 369–381. Springer Berlin / Heidelberg, 2000.

[16] Astro Teller and David Andre. Automatically choosing the number of fitness cases: The rational allocation of trials. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 321–328, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[17] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of the Scientific Discovery through Advanced Computing Conference*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[18] Richard Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *ACM/IEEE Conference on Supercomputing*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.