Unbounded Differential Evolution with Very Large Populations

Xiuyi Guo
The University of Tokyo
Meguro, Japan
ORCID 0009-0000-5430-1077

Tomofumi Kitamura
The University of Tokyo
Meguro, Japan
ORCID 0009-0008-4272-3641

Alex Fukunaga
The University of Tokyo
Meguro, Japan
ORCID 0009-0007-9960-5358

***Abstract*—Standard evolutionary search algorithms, including differential evolution (DE), use a relatively small, limited-size population that remains the same size or gradually shrinks during the search. It has long been believed that replacement (discarding some members of the population and inserting new candidate individuals in their place) is an important feature of evolutionary search. Recent work on Unbounded DE (UDE) has shown that non-increasing populations are not necessary, and that with an appropriate selection operator, a DE with a monotonically growing population can perform comparably to state-of-the-art adaptive DE. However, UDE has scalability issues, as the runtime complexity of each iteration increases with population size, and limited RAM capacity poses a limitation on the growth of the population. We propose extensions to UDE that overcome these limitations by (1) efficiently storing the population on external storage (SSD) and (2) reducing the computational complexity of tournament selection. We show experimentally that this allows UDE to scale to very large populations with 2×10^8 individuals.**

***Index Terms*—differential evolution, population size**

I. Introduction

The notion of a limited-size population, where some or all members of the population are *replaced* by newly generated individuals, has been a common component of evolutionary search algorithms. However, removing members from the population discards possibly valuable information from the search process. In order to enhance search performance, some evolutionary search algorithms have used *auxiliary* populations, e.g., *archives*, which are a subset of previously generated individuals that are no longer in the main population [1].

In order to establish a simpler, more uniform framework that does not require auxiliary population structures, recent work has proposed Unbounded DE (UDE) [2], [3]. UDE keeps all generated individuals and does not discard any individuals, relying on selection to focus the search. It was shown that USHADE, a variant of UDE which incorporates success-history based control parameter adaptation, is competitive with standard adaptive DEs such as SHADE [4] and LSHADE [5].

However, storing all generated individuals in a monotonically growing population poses scaling challenges for UDE. First, the memory required to store the population can exceed the amount of RAM available. For example, consider the widely used CEC2013LSGO [6] benchmarks. The standard problem size is 1000 dimensions, so each individual vector requires 1000 double precision (8-byte) floats per individual (= 8KB/individual). A very long UDE search run to 2×10^8 individuals would require 1.6TB, far exceeding the RAM available in a standard machine. Second, the straightforward implementation of tournament selection required at each iteration of the algorithm scales linearly with population size, resulting in a significantly lower search rate (individual evaluation rate) than traditional DE.

This paper addresses both of these limitations. First, we propose a method for decoupling the storage of the individual fitness values and individual vectors, so that only the individual fitness values (usually ≤ 8 bytes) are kept in RAM, while the individual vectors (which can be much larger) are stored in secondary storage (i.e., SSD). Second, we propose a tournament selection algorithm which uses an order-statistics tree (OST) [7] to reduce the complexity of tournament selection from $O(|P|)$ to $O(\log |P|)$, where P is the population size. We show that a multi-threaded parallel implementation of UDE which uses these enhancements can achieve individual generation rates approaching those of parallel implementations of RAM-based DE even with extremely large populations ($P > 10^8$), and that search performance is also competitive with RAM-based DE.

The rest of the paper is organized as follows. Section II reviews related work on UDE. Section III proposes PUDE/SSD, which overcomes the scaling issues of UDE by using external storage (SSD) for individual vectors, and speeding up tournament selection using an OST. We also introduce PUSHADE/SSD, which incorporates these enhancements to UDE with control parameter adaptation. Section IV experimentally evaluates the proposed methods. We show that the proposed methods allow UDE to scale successfully to extremely large populations (2×10^8). Section V concludes with a summary and discussion.

II. Background

Differential Evolution (DE) [8] is a population-based stochastic optimization algorithm that maintains a population set P, where each individual represents a candidate solution to a fitness function f. Each solution is a D-dimensional vector. The population size is denoted by $|P^t|$, where t represents the current generation number. At each generation, new individuals are generated by first selecting parent vectors from the population, and then applying mutation/crossover

operators to the parents. The fitnesses of the new individuals are evaluated, and individuals that are superior to their parents will replace their parents in the population; otherwise, they are discarded.

Population size is a key control parameter that has a significant impact on the performance of differential evolution algorithms. A larger population means a more diverse composition of individuals, which facilitates a broader exploration of the search space. Typically, a high-dimensional problem, such as one with 1000 dimensions, requires a considerably large population [5].

In addition to this basic framework, state-of-the-art DE variants such as SHADE [4] incorporate control parameter adaptation mechanisms that significantly improve search performance by dynamically modifying parameters such as mutation rate and crossover rate during search [9]. Population size reduction has been shown to be effective in improving search performance, especially when the evaluation budget (maximum number of evaluations) for search is known a priori and an appropriate reduction schedule is applied, e.g., LSHADE reduces population size according to a linear schedule [5].

A. Unbounded DE (UDE)

Unlike standard DE, Unbounded DE (UDE) [2], [3] stores all generated offspring in P^{t+1}. In UDE, an offspring set Q of size A is generated each time. New offspring vectors are *appended* to the population, and parents are never discarded, i.e., there is no replacement. Parent vectors are selected using a tournament strategy (see below) instead of the uniform random sampling used in standard DE. UDE can be combined with standard enhancements for DE, including various mutation and crossover operators and adaptive parameter control. USHADE, which incorporates current-to-pbest mutation strategy [1] and success-history based control parameter adaptation [4] has been shown to perform comparably to SHADE and LSHADE [3].

B. Selection in UDE

a) Simple Tournament Selection (T): Let $P^t = \{x^1, \ldots, x^{|P^t|}\}$ denote the population at iteration t. Simple tournament selection (denoted as "T" in the rest of the paper) selects a parent by first sampling $n = \left\lceil \frac{|P^t|}{T} \right\rceil$ individuals uniformly at random from P^t, where $T > 0$ controls the selection pressure. The sampled individual with the best fitness value is selected. Smaller values of T correspond to stronger selection pressure, while larger values of T promote weaker selection and increased diversity.

b) Diversity-Preserving Tournament Selection (DPT): DPT seeks to improve upon standard tournament selection by restricting tournaments to structured subpopulations. Each individual in P^t is assigned an insertion index, and a fixed generation size A is defined. For each selection event, an index $j \sim \mathcal{U}\{1, \ldots, A\}$ is sampled, and a subset $S^j = \{x^i \in P^t \mid i \equiv j \pmod{A}\}$ is constructed. Tournament selection with parameter T is then applied within S^j. By limiting competition to individuals with similar generational structure, DPT reduces repeated selection of the same elite lineages and helps preserve population diversity.

III. PUDE/SSD: Improving the Scalability of UDE

To address the scalability limitations of UDE, we propose Parallel UDE with SSD-based populations (PUDE/SSD). Algorithm 1 shows PUDE/SSD. The two main novel contributions of PUDE/SSD are (1) using external storage (SSD) to store individual vectors, and (2) a more efficient approach to implementing tournament selection which reduces the computational complexity to $\mathrm{O}(\log |P|)$ per selection (vs. $\mathrm{O}(|P|)$ for standard UDE tournament selection). In addition, PUDE incorporates a straightforward multithreaded parallelization of UDE. Each of these is described below.

A. Parallelization

Although parallelization is straightforward and orthogonal to the main contributions of the paper described below, we first parallelize UDE, for 3 reasons: (1) since we are trying to scale UDE to population sizes large enough to exhaust RAM, it does not make sense to leave CPU cores idle (as there will not be enough RAM available to perform other tasks with the other cores), (2) we want to show the feasibility of SSD-based UDE under the *maximum* load to the SSDs (this occurs when many threads are simultaneously accessing the SSD; a single-threaded implementation would not be an appropriate stress test for modern NVMe SSDs), and (3) experimental runs with very large populations such as in Section IV would require too much time with a standard single-threaded implementation so parallelization is required to speed up the experiments.

The parallelization is straightforward and implemented in Alg. 1, line 8. Batches of offspring of size A are processed in parallel, as there are no dependencies.

B. Storing the Population on External Storage (SSD)

If the memory required to store the population exceeds available RAM, external storage such as SSDs can be used to augment RAM. Current consumer-grade SSDs with an NVMe interface support 2×10^6 random read/write operations per second using deep, parallel I/O queues. However, random read/writes are still much slower than RAM. A trivial approach to handling extremely large populations exceeding RAM capacity would be to simply implement UDE as originally specified in [3] and rely on the virtual memory handler of the operating system to swap the necessary data in/out of RAM. This trivial approach results in extremely slow performance. The reason is that tournament selection requires random sampling of many individuals from the population *uniformly*. This has no locality (due to uniform sampling), so almost every access to an individual results in a page fault, requiring accessing the SSD instead of RAM.

Another straightforward approach is to store the population as an SSD file, where the file stores a contiguous sequence of individuals (the vector data for the individuals as well as their fitness values), and individual structures are read into RAM as

needed. As with the previous virtual memory-based approach, this performs poorly due to a lack of locality in tournament selection (instead of page faults, there are cache misses as individuals are constantly loaded and displaced from the disk cache).

A significant speedup can be obtained by exploiting the fact that tournament selection is only based on the fitness values of the individuals. The individual vector values are only necessary for the individuals that win the tournaments. PUDE/SSD uses offset-based I/O for individuals. By storing all of the individual fitnesses in an array in RAM, and storing their corresponding individual vectors to a file on the SSD (randomly accessed using `pread` and `pwrite` with offsets), we can reduce the number of SSD accesses to 4 random reads (for $pbest$, p, r_1, r_2 in line 11) and one write (for the new $u^{i,t}$ in line 15).

Algorithm 1 PUDE/SSD (red is difference from UDE [3])

1: $t \leftarrow 1$;
2: Initialize population offset set P^t;
3: **for** $i \leftarrow 1$ **to** $|P^t|$ **do**
4: Evaluate $f(x^{i,t})$;
5: Write $x^{i,t}$ to SSD and save offset($x^{i,t}$) in P^t;
6: **while** not termination condition **do**
7: Initialize offspring offset set Q of size A;
8: **for** $i \leftarrow 1$ **to** A **in parallel** **do**
9: Select $pbest$;
10: Select p, r_1, r_2 $(p \neq r_1, p \neq r_2, r_1 \neq r_2)$;
11: Read $x^{pbest,t}, x^{p,t}, x^{r_1,t}, x^{r_2,t}$ from SSD; ▷ with `pread`
12: Generate $v^{i,t}$ using mutation;
13: Generate $u^{i,t}$ using crossover;
14: Evaluate $f(u^{i,t})$;
15: Write $u^{i,t}$ to SSD; ▷ with `pwrite`
16: $Q^i \leftarrow$ offset($u^{i,t}$);
17: $P^{t+1} \leftarrow P^t \cup Q$;
18: $t \leftarrow t + 1$;

The `pread`/`pwrite` system calls perform I/O to a file descriptor (fd) with explicitly specified offset and size. In PUDE/SSD, explicit offset management guarantees that read/write segments do not overlap, so `pread` and `pwrite` can be safely executed asynchronously for the same fd.

C. Tournament Selection Optimization

Tournament selection (Section II-B) in standard UDE has a worst-case time complexity of $O(|P|)$. Although an adaptive parameter T in USHADE controls the pressure of the tournament, the selection process does not become a significant bottleneck only when the actual number of samples is less than 10^4. However, when the population size is extremely large ($|P| \sim 10^8$), the actual sample size can easily reach 10^6. This means that tournament selection can become the dominant computational bottleneck (even more than fitness evaluation).

1) Order Statistic Tree-based Tournament Selection (OST-T): Algorithm 2 shows OST-T, which significantly speeds up the tournament selection process, with a time complexity of $O(\log |P|)$.

Algorithm 2 OST-T

Require: OST $\mathcal{T}$ for key $f(x)$ and value x, the number n of samples;
Ensure: the selected individual x^*;
1: Sample $U \sim \text{Uniform}(0, 1)$
2: $K \leftarrow \lceil |P| - |P|(1-U)^{\frac{1}{n}} \rceil$
3: $x^* \leftarrow \text{value}(\text{Select}(\mathcal{T}, K))$ ▷ OST selection
4: **return** x^*

Let population $P = \{x^1, x^2, \ldots, x^{|P|}\}$ be a dynamic set of $|P|$ individuals. Tournament selection (Section II-B) draws n samples from P and returns the individual with minimum fitness $f(x)$.

OST-T efficiently samples the selected element without explicitly sampling from P, while supporting dynamic updates to P.

a) Select a rank K in $|P|$.: Let random variable K be the minimum rank of fitness in the n draws.

For any integer $k \in \{1, \ldots, |P|\}$, the cumulative distribution function (CDF) of K can be derived as follows. The event $\{K \leq k\}$ is equivalent to the complement of $\{K > k\}$, where $\{K > k\} \iff$ none of the elements $\{1, 2, \ldots, k\}$ appears in the n draws.

Since each draw avoids $\{x_1, \ldots, x_k\}$ with probability $\frac{(|P|-k)}{|P|}$, and the draws are independent, we obtain $\Pr(K > k) = \left(\frac{|P|-k}{|P|}\right)^n$. Therefore, the CDF of K is $F(k) := \Pr(K \leq k) = 1 - \left(\frac{|P|-k}{|P|}\right)^n$.

Sampling from this distribution can be performed efficiently via inverse transform sampling. Let $U \sim \text{Uniform}(0, 1)$. Setting $U = F(K)$, we obtain: $U = 1 - \left(\frac{|P|-K}{|P|}\right)^n$, $K = |P| - |P|(1-U)^{\frac{1}{n}}$.

Since K must take integer values in $\{1, \ldots, |P|\}$, the sampled value is obtained by applying an appropriate integer rounding operation and clamping the result to the valid range if necessary. For example, fix K with the ceiling function, then $K^* = \lceil K \rceil$ will be considered as K.

This procedure requires only constant-time arithmetic operations and thus samples K with a time complexity of $O(1)$.

b) Select the individual x^* in P.: An *Order-Statistics Tree* (OST) is an augmented balanced binary search tree that supports efficient order-based queries in addition to standard dynamic set operations [7]. Typical implementations are based on red-black trees or AVL trees. An OST supports a Select($\mathcal{T}$,k) operation that returns the element with the k-th smallest key in the OST $\mathcal{T}$ with time complexity $O(\log |P|)$.

We maintain an OST $\mathcal{T}$ which has key-value pairs $(f(x), x)$, where the keys are the fitness $f(x)$ of individuals x and the values are the pointers to x. Since the OST supports dynamic insertion, we can update $\mathcal{T}$ after each generation t. The time complexity of inserting each offspring is $O(\log |P|)$.

After sampling the rank $K \in \{1, \ldots, |P|\}$ according to the distribution derived above, we first find the corresponding value based on the key $f(x^*)$ with rank K, and then select the corresponding individual x^* from P as $x^* = \text{value}(\text{Select}(\mathcal{T}, K))$.

Therefore, the total time complexity of each tournament is $\mathrm{O}(\log |P|)$.

For OST-based Diversity-Preserving Tournaments (OST-DPT) (Section II-B), we need to maintain a set of OSTs with size A and select x^* from the corresponding OST. The remaining steps are exactly the same as for OST-T, and the overall complexity is the same $\mathrm{O}(\log |P|)$.

D. PUSHADE/SSD: Integrating the proposed techniques into an adaptive UDE

Current state-of-the-art DE methods all incorporate some parameter adaptation mechanism [9]. Previous work showed that USHADE (an adaptive variant of UDE using success-history based adaptation [4]) significantly outperforms UDE [3]. Algorithm 3 shows PUSHADE/SSD, which incorporates our proposed methods (SSD storage of individuals, OST-T, and parallelism) into USHADE.

Algorithm 3 PUSHADE(T)/SSD (red is difference from USHADE [3])

$t \leftarrow 1$;
$k \leftarrow 0$;
Initialize population offset set P^t;
Init. success histories $M_F[1 \ldots H], M_C[1 \ldots H] \leftarrow 0.5$;
Initialize success history $M_T[1 \ldots H] \leftarrow |P^1|$;
Initialize order-statistics tree $\mathcal{T}$;
for $i \leftarrow 1$ **to** $|P^t|$ **do**
    Evaluate $f(x^{i,t})$;
    Insert $f(x^{i,t})$ in $\mathcal{T}$;
    Write $x^{i,t}$ to SSD and save offset$(x^{i,t})$ in P^t;
while not termination condition **do**
    Initialize offspring offset set Q of size A;
    $S_F \leftarrow \varnothing$, $S_C \leftarrow \varnothing$, $S_{\Delta f} \leftarrow \varnothing$;
    $S_T \leftarrow \varnothing$;
    for $i \leftarrow 1$ **to** A **in parallel do**
        Randomly select $r^i \in \{1, \ldots, H\}$;
        while $F^{i,t} \leq 0$ **do**
            $F^{i,t} \leftarrow \min(\text{rand}_{\text{Cauchy}}(M_F[r^i], 0.1), 1)$;
        $C^{i,t} \leftarrow \max(\min(\text{rand}_{\text{Normal}}(M_C[r^i], 0.1), 1), 0)$;
        $T^{i,t} \leftarrow \max(\text{rand}_{\text{Normal}}(M_T[r^i], 10), 100)$;
        Select *pbest*;
        Select p, r_1, r_2 $(p \neq r_1, p \neq r_2, r_1 \neq r_2)$ from $\mathcal{T}$ using OST-T;
        Read $x^{pbest,t}, x^{p,t}, x^{r_1,t}, x^{r_2,t}$ from SSD; ▷ with `pread`
        Generate $v^{i,t}$ using *current-to-pbest/1* mutation;
        Generate $u^{i,t}$ using *binomial* crossover;
        Evaluate $f(u^{i,t})$;
        Write $u^{i,t}$ to SSD; ▷ with `pwrite`
        $Q^i \leftarrow$ offset$(u^{i,t})$;
    for $i \leftarrow 1$ **to** A **do** ▷ sequential update
        Insert $f(u^{i,t})$ in $\mathcal{T}$;
        if $f(u^{i,t}) \leq f(x^{p,t})$ **then**
            Append $F^{i,t}$ to S_F, $C^{i,t}$ to S_C, and $f(x^{p,t}) - f(u^{i,t})$ to $S_{\Delta f}$;
            Append $T^{i,t}$ to S_T;
    $P^{t+1} \leftarrow P^t \cup Q$;
    if $S_F \neq \varnothing$ and $S_C \neq \varnothing$ **then**
        $M_F[k] \leftarrow \text{mean}_L(S_F, S_{\Delta f})$;
        $M_C[k] \leftarrow \text{mean}_L(S_C, S_{\Delta f})$;
        $M_T[k] \leftarrow \text{mean}_L(S_T, S_{\Delta f})$;
        $k \leftarrow (k+1) \bmod H$;
    $t \leftarrow t + 1$;

IV. Experimental results

We experimentally evaluate our proposed methods using the widely used IEEE CEC2013 Large-Scale Global Optimization (LSGO) benchmarks. Although recent IEEE CEC LSGO competitions have introduced newer benchmarks, we use the CEC2013 benchmarks because this paper focuses on very large populations with high memory usage, and the high dimensionality of the CEC2013 benchmarks (1000) requires more memory to store individual vectors than more recent benchmarks. In addition, CEC2013 functions are relatively fast to evaluate, compared to more recent functions which require more computation per evaluation – fast evaluation functions pose the *most challenging/disadvantageous* conditions for SSD-based UDE, because of the comparatively high percentage of time spent outside of the evaluation (such as accessing SSD to read/write individual vector data).

All experiments were run on an AMD Ryzen 9 9950X3D (16 cores) with 96GB RAM and a Samsung 990 Pro 2TB SSD (NVMe Gen4) running Ubuntu 25.10. All algorithms were implemented in C++ (g++ with -O3 and -std=c++23 compilation flags).

A. Evaluation of OST-T

We first confirm that OST-T has stochastic behavior very similar to the naive implementation of tournaments (Naive-T). We generate a population of size $|P| = 10^6$. For each trial, we uniformly sample n elements from the population and record the sampled element's rank. This sampling procedure is repeated independently for 10^7 trials to obtain statistically stable estimates of the empirical distribution. The cumulative value of the sampled ranks is then computed. This experiment is conducted for $n \in \{1, 10000\}$, because Naive-T is prohibitively slow when n is larger. In all cases, a Kolmogorov-Smirnov test showed no significant differences between the Naive-T and OST-T distributions.

Next, we compare the runtime of OST-T vs. Naive-T. For $n = 10000$, Naive-T required 14637ms, while OST-T required 73ms. This clearly shows that OST-T, which runs in logarithmic time, scales much better than Naive-T, which

runs in linear time.

B. Evaluation of PUSHADE/SSD: Experimental Settings

We compare the following algorithms on the CEC2013LSGO benchmarks ($D = 1000$ dimensions), 10 runs of 2e8 evaluations/run, all using 16 threads.

- PUSHADE(T)/SSD (PUSHADE/SSD using the OST-based simple tournament T), initial $|P| = 18 \times D$, memory size H = 6, p-best rate p = 0.11.
- PUSHADE(DPT)/SSD (PUSHADE/SSD using the OST-based Diversity-Preserving Tournament DPT, initial $|P| = 18 \times D$, memory size H = 6, p-best rate p = 0.11.
- PSHADE, $|P| = 100$, memory size 30, archive rate r_{arc} = 2.0, p-best rate p = 0.10.
- PLSHADE, initial $|P| = 18 \times D$, memory size H = 5, archive rate r_{arc} = 1.4, p-best rate p = 0.11.

C. Evaluation Rate

Table I compares the evaluation rates on 16 threads, average of 10 runs with evaluation budget $E_{\max} = 2 \times 10^8$/run. The evaluation rate of PUSHADE(DPT)/SSD is in thousands of evaluations/second. The other algorithms are reported as rates relative to PUSHADE(DPT)/SSD (< 1 is slower than PUSHADE(DPT)/SSD, > 1 is faster, e.g., on function f_1, PSHADE eval rate is 1.34 faster than PUSHADE(DPT)/SSD).

Although PSHADE and PLSHADE operate entirely in RAM, their performance advantage over PUSHADE(DPT)/SSD is relatively modest. For the majority of the benchmark functions (11 out of 15, specifically f_1–f_6, f_8–f_{11}, and f_{13}), PLSHADE is less than $1.50\times$ faster (i.e., less than a 50% advantage) compared to PUSHADE(DPT)/SSD. This indicates that the additional latency introduced by SSD I/O is effectively mitigated, keeping the evaluation rate within a comparable range.

PUSHADE(T)/SSD has a slightly lower evaluation rate than PUSHADE(DPT)/SSD on most functions. In these experiments, the OST maintained by the T variant is approximately two orders of magnitude larger than that of DPT. Consequently, T incurs higher maintenance and traversal costs during selection and update operations, on top of the SSD access overhead. This increased combined cost directly translates into a lower evaluation rate for PUSHADE(T)/SSD compared to PUSHADE(DPT)/SSD across most benchmark functions.

1) Ablation Study: In order to identify the contribution of each of the ideas proposed in this section to speeding up the evaluation rate, we performed an ablation study. For a representative function (f_4), we compare variants of PUSHADE, each executed once with $E_{max} = 5 \times 10^7$. This is a smaller scale experiment than the other experiments in this section due to the very low evaluation rates of some of the variants tested.

We evaluate 8 variants of PUSHADE, each using a different combination of (a) tournament implementation ("naive-T", "T", and "DPT", where "T" and "DPT" are OST-based implementations), and (b) population storage strategies ("RAM": implementation of parallel USHADE which stores all individuals in RAM and terminates when RAM is exhausted, "Unified-VM": stores SSD array of individual structs containing both fitness and vector values, "SSD": proposed decoupled storage strategy which stores individual fitness values in RAM and individual vector values on SSD). For example, PUSHADE(naive-T)/Unified-VM is a variant of PUSHADE which uses the naive tournament implementation from [3], and stores an array of individual structs (fitness and vector values) on SSD, while PUSHADE(T)/SSD uses OST-based simple tournaments and uses the proposed decoupled SSD storage strategy.

TABLE I
COMPARISON OF EVALUATION RATE (16 THREADS)

func.	PLSHADE	PSHADE	PUSHADE (T)/SSD	PUSHADE (DPT)/SSD
1	1.37×	1.34×	0.91×	1.4017×10^5
2	1.12×	1.34×	0.79×	1.6838×10^5
3	1.40×	1.77×	0.92×	1.3095×10^5
4	1.31×	1.32×	0.89×	1.4653×10^5
5	1.25×	1.53×	0.92×	1.4221×10^5
6	1.41×	1.52×	0.89×	1.2329×10^5
7	2.34×	2.40×	0.88×	1.4966×10^5
8	1.45×	1.48×	0.89×	1.2231×10^5
9	1.25×	1.25×	0.95×	1.2540×10^5
10	1.38×	1.33×	1.07×	1.0478×10^5
11	1.46×	1.59×	0.94×	1.1983×10^5
12	3.53×	4.59×	0.88×	1.4693×10^5
13	1.39×	1.52×	0.88×	1.2587×10^5
14	1.52×	1.50×	0.89×	1.2658×10^5
15	2.40×	2.63×	1.20×	9.9720×10^4

Figure 1 plots cumulative evaluation rate (total number of evaluations so far / runtime so far) vs. the number of evaluations for each variant. The cumulative evaluation rate changes as search progresses because the evaluation rate at any given time depends on the current state of the data structures (e.g., the OST) and system (e.g., frequency of SSD cache misses). The results show:

- The variants using naive-T tournament implementation start with a fast rate, as naive-T is efficient for small population size $|P|$, but drastically slows down as $|P|$ increases, showing the significant advantage of OST-based tournament implementation.
- The variants using RAM only storage are initially slightly faster than the corresponding variants using Unified-VM and SSD, but they terminate prematurely at approx. 1.1×10^7 evals due to RAM exhaustion.
- The decoupled SSD variants are faster than the corresponding Unified-VM variants, showing the advantage of decoupled SSD storage.

Overall, we see that combining both decoupled SSD storage of individual vectors and OST-based tournament selection yields

the highest evaluation rate.

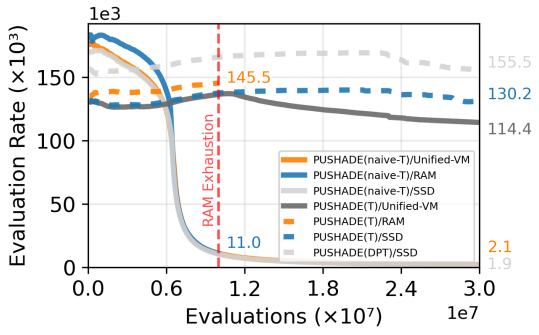

Fig. 1. Ablation Study: Evaluation Rate Comparison of PUSHADE Variants

D. Search Performance

We evaluate search performance in two ways. First, we compare the search performance vs. the number of fitness evaluations, to validate whether the UDE-based PUSHADE/SSD algorithm has competitive search behavior (independent of the evaluation rate) for runs with extremely large number of fitness evaluations. Next, we evaluate the search performance vs. runtime. This incorporates the effect of the gap in evaluation rates between the RAM-based algorithms (PSHADE, PLSHADE) and the SSD-based PUSHADE/SSD algorithms.

1) Search Performance vs. Number of Evaluations: Table III compares search performance at E=3 $\times$ 10^6, 1 $\times$ 10^7, 5 $\times$ 10^7, and 2 $\times$ 10^8 evaluations. The symbols indicate statistically significant difference relative to PUSHADE(DPT)/SSD, according to Wilcoxon rank-sum test ($p < 0.05$): "$+$" indicates "significantly lower (better) f achieved compared to PUSHADE(DPT)/SSD, "$-$" indicates "significantly higher (worse) f achieved compared to PUSHADE(DPT)/SSD, and "$\approx$" indicates "f not significantly different than PUSHADE(DPT)/SSD".

Comparing these results, we observe:

- PSHADE has the best overall performance at $E = 3 \times 10^6$.
- PUSHADE(DPT)/SSD has the best overall performance at $E = 1 \times 10^7$ and $E = 5 \times 10^7$.
- PLSHADE performed best at $E = 2 \times 10^8$.

These results are consistent with previously reported results that compared USHADE to SHADE and LSHADE. The defining feature of LSHADE and PLSHADE (compared to SHADE and PSHADE, respectively) is the use of a linear population reduction schedule. The performance of LSHADE and PLSHADE depends crucially on a linear population schedule, where the evaluation budget $E_{\max}$ is assumed to be known, and the population size $|P|$ is reduced linearly such that $|P| = |P|_{min}$ at $E_{\max}$.

However, in many practical applications, we do *not* know a priori when the search algorithm will be terminated, i.e., $E_{\max}$ is unknown, so the algorithm must return the best solution available when the algorithm is terminated (e.g., the user/engineer is instructed, "submit the best solution you have right now").

Thus, if we are not given a precise $E_{\max}$ budget ahead of time, then algorithms such as SHADE, USHADE, and USHADE/SSD which do not depend on prior knowledge of $E_{\max}$ are preferable.

Overall, we observe that the performance of PUSHADE/SSD relative to PSHADE and PLSHADE follow the same trends previously reported for USHADE vs. SHADE and LSHADE, i.e., the results show that the search behavior of UDE-based algorithms scales successfully even for very large populations.

2) Search Performance vs. Runtime (Does SSD-based storage result in a significant performance penalty?): Figure 2 evaluates search performance (best-so-far fitness score) vs. runtime. Table II compares search performance at various times, where time for each evaluation milestone is defined as the time when the earliest function finished that number of evaluations across all algorithms, i.e., $T(E)$ = time when earliest function finished E evals. The symbols indicate statistically significant difference relative to PUSHADE(DPT)/SSD, according to Wilcoxon rank-sum test ($p < 0.05$): "$+$" indicates "significantly lower (better) f achieved compared to PUSHADE(DPT)/SSD, "$-$" indicates "significantly higher (worse) f achieved compared to PUSHADE(DPT)/SSD, and "$\approx$" indicates "f not significantly different than PUSHADE(DPT)/SSD".

Unlike the previous comparison of search performance vs. number of evaluations, this view of the data incorporates all runtime overheads caused by algorithm components in PUDE/SSD. Unlike PSHADE and PLSHADE, where all operators have $O(1)$ computational complexity, PUSHADE(T)/SSD and PUSHADE(DPT)/SSD use a tournament operator with $O(\log |P|)$ computational cost. Also, unlike PSHADE and PLSHADE which only use RAM, the PUSHADE/SSD variants use SSD to store individual vectors and incur latency each time the SSD is accessed. Therefore, as shown above in Section IV-C, PUSHADE/SSD generally performs fewer evaluations per unit time given the same CPU resources.

Despite this inherent disadvantage in raw evaluation rate, Figure 2 shows that PUSHADE/SSD achieves superior search performance compared to PSHADE with a fixed runtime budget. PUSHADE(DPT)/SSD consistently attains better solution quality than PSHADE on a substantial subset of benchmark functions, indicating that it makes more effective use of the available time budget, despite the storage access overheads due to the SSD.

3) Comparison with state-of-the-art, DE+Local Search hybrid algorithm for LSGO: SHADE-ILS [10] is a state-of-the-art algorithm for LSGO, which combines two components: (a) an exploratory component, which is SHADE, and (b) an intensification local search component. At each *iteration* consisting of I fitness evaluations, SHADE-ILS executes SHADE for I_S evaluations, and then executes local search for $I - I_S$ evaluations, alternating between exploration and intensification

(in [10], $I = 50,000$ and $I_s = 25,000$). SHADE-ILS has consistently performed well in CEC LSGO competitions.

Note that the PUSHADE variants are not designed to be complete, state-of-the-art methods for LSGO. *PUSHADE can be considered a potential replacement for the SHADE component of an exploration+intensification hybrid algorithm such as SHADE-ILS*. Nevertheless, it is interesting to compare PUSHADE vs SHADE-ILS on very long runs.

Table III shows that while SHADE-ILS significantly outperforms the PUSHADE variants at E=3 $\times$ 10^6, the performance gap between the SHADE-ILS and PUSHADE(DPT)/SSD decreases as E increases, and by E=2 $\times$ 10^8, PUSHADE(DPT)/SSD performs comparably to SHADE-ILS, outperforming SHADE-ILS on 6/15 problems, while SHADE-ILS outperforms PUSHADE(DPT)/SSD on 7/15 problems.

Note that we do *not* include SHADE-ILS in the runtime-based comparison in Table II because SHADE-ILS is a serial algorithm and it would not be fair to compare the runtimes of a serial algorithm with our parallel algorithms (parallelizing a hybrid metaheuristic such as SHADE-ILS is beyond the scope of this paper).

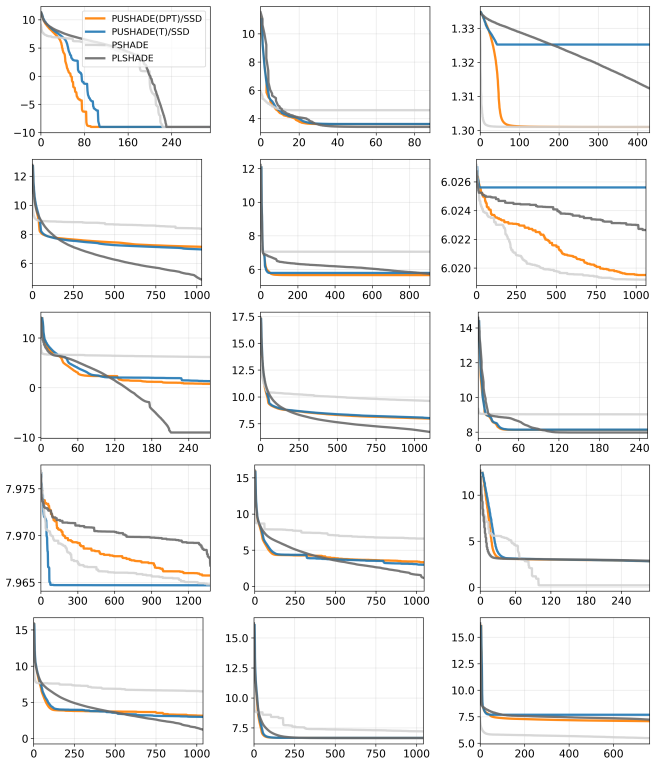

Fig. 2. Search performance on benchmark functions $f_1, ..., f_{15}$ (arranged left to right, top to bottom, e.g., first row shows f_1 to f_3). All curves are averages of 10 independent runs for each function (x-axis: runtime t, y-axis: $\log_{10}(f(x)_{\text{best}})$, where $f(x)_{\text{best}}$ is best-so-far fitness).

V. Conclusions and Future Work

To overcome the scaling issues of UDE with very large populations, we proposed SSD-based storage of individual vectors and faster tournament selection based on OSTs. We showed that with these enhancements, parallel UDE using SSD storage can achieve high evaluation rates comparable to parallel RAM-based DEs such as SHADE and LSHADE. Furthermore, we showed that the search efficiency of PUDE/SSD is comparable to parallel SHADE and LSHADE. Overall, these results showed that these enhancements allow UDE to scale to very large populations. Practical performance is achievable with a search framework that never discards any individuals from the population, even if the population grows to over 10^8 individuals under the most challenging conditions (benchmarks with fast evaluation functions and multithreaded search that magnify the effects of SSD latency).

While we showed that search with extremely large population sizes can be competitive with standard algorithms such as SHADE, we do *not* claim that UDE-based approaches achieve new, state-of-the-art search performance, as that remains a task for future work. Now that we have proposed techniques to enable fast evaluation with massive populations (OST-T and SSD-based individual storage), one promising direction is improving the UDE search algorithm (e.g., improving the selection policy). Another promising direction to achieve state-of-the-art performance for LSGO problems is to investigate hybrid DE+local search algorithms similar to the state-of-the-art SHADE-ILS, where the SHADE component is replaced by USHADE.

The proposed OST-based selection mechanism is not limited to tournament selection. The core idea of sampling individuals by rank according to a known probability distribution can be generalized to support a wide range of selection schemes. In principle, any selection strategy that can be expressed as a probability distribution over population ranks (e.g., exponential ranking, linear ranking, truncated distributions, or hybrid adaptive schemes) can be efficiently implemented on top of an OST, making OST-based selection widely applicable for large-scale evolutionary search.

The performance gap between SSD-based and standard RAM-based DE is expected to shrink further with ongoing advances in storage hardware. State-of-the-art NVMe Gen5 SSDs (e.g., Samsung 9100 Pro [11]) already achieve random I/O performance exceeding 2200K IOPS on 4KB random I/O (twice as fast as the NVMe Gen4 Samsung 990 used in our experiments). As storage bandwidth, queue depth, and parallelism continue to improve, SSD-based population storage will increasingly resemble an extension of main memory rather than a performance bottleneck. Future work will seek to further optimize UDE to exploit the memory hierarchy in order to close the evaluation rate gap with RAM-based search.

References

[1] J. Zhang and A.C. Sanderson. Jade: Adaptive differential evolution with optional external archive. *IEEE Transactions on Evolutionary Computation*, 13:945–958, 2009.

[2] T. Kitamura and A. Fukunaga. Differential evolution with an unbounded population. In *2022 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2022.

[3] T. Kitamura and A. Fukunaga. Is selection all you need in differential evolution? *Applied Soft Computing*, 186:114138, 2026.

TABLE II
COMPARISON OF SEARCH PERFORMANCE AT VARIOUS TIMES

$T(3\times10^6)$					$T(1\times10^7)$					$T(5\times10^7)$					$T(2\times10^8)$				
Function	**PLSHADE**	**PSHADE**	**PUSHADE(T)/SSD**	**PUSHADE(DPT)/SSD**	**Function**	**PLSHADE**	**PSHADE**	**PUSHADE(T)/SSD**	**PUSHADE(DPT)/SSD**	**Function**	**PLSHADE**	**PSHADE**	**PUSHADE(T)/SSD**	**PUSHADE(DPT)/SSD**	**Function**	**PLSHADE**	**PSHADE**	**PUSHADE(T)/SSD**	**PUSHADE(DPT)/SSD**
1	+	+	−	9.6021e+09	1	−	+	−	1.7889e+08	1	−	−	−	0.0000e+00	1	≈	≈	≈	0.0000e+00
2	−	+	≈	2.3450e+05	2	−	−	−	1.0005e+04	2	+	−	≈	4.1529e+03	2	+	−	≈	4.1529e+03
3	≈	+	≈	2.1577e+01	3	−	+	+	2.1409e+01	3	−	+	−	2.0010e+01	3	−	+	−	2.0000e+01
4	+	+	≈	2.6104e+12	4	+	+	−	1.2732e+10	4	−	−	≈	7.6188e+07	4	+	−	≈	4.0507e+07
5	+	+	≈	9.2839e+11	5	−	−	−	7.6445e+06	5	−	−	−	4.7966e+05	5	≈	−	−	4.6012e+05
6	+	≈	≈	1.0635e+06	6	+	+	≈	1.0611e+06	6	−	≈	−	1.0565e+06	6	−	+	−	1.0543e+06
7	+	+	−	4.8103e+12	7	+	+	−	1.4107e+07	7	−	−	≈	2.1755e+02	7	+	−	−	6.4593e+00
8	+	+	≈	6.9801e+16	8	−	+	−	1.1403e+12	8	−	−	≈	1.1160e+09	8	+	−	≈	3.7725e+08
9	−	+	−	9.2738e+11	9	−	−	≈	6.2567e+08	9	+	−	≈	1.3554e+08	9	+	−	≈	1.3553e+08
10	+	≈	≈	9.4594e+07	10	≈	≈	≈	9.4143e+07	10	≈	+	≈	9.3797e+07	10	−	+	≈	9.3174e+07
11	+	+	≈	1.2619e+15	11	−	+	−	1.8221e+09	11	−	−	−	8.0448e+04	11	−	−	≈	1.8568e+04
12	+	+	≈	1.3919e+12	12	+	≈	−	6.0246e+05	12	−	+	−	1.0379e+03	12	−	+	≈	6.3838e+02
13	+	+	≈	9.9747e+14	13	≈	+	−	3.5825e+09	13	−	−	−	2.6892e+04	13	−	−	≈	5.9685e+03
14	+	+	≈	1.4080e+15	14	−	+	−	1.0194e+10	14	−	−	−	5.0738e+06	14	≈	−	≈	4.6945e+06
15	+	+	+	1.0285e+16	15	−	+	−	1.1908e+08	15	≈	+	≈	2.4118e+07	15	≈	+	≈	1.6710e+07

TABLE III
COMPARISON OF SEARCH PERFORMANCE AT 3×10^6, 1×10^7, 5×10^7, 2×10^8 EVALUATIONS

(3×10^6)						(1×10^7)						(5×10^7)						(2×10^8)					
Function	**PLSHADE**	**PSHADE**	**PUSHADE(T)/SSD**	**SHADE-ILS**	**PUSHADE(DPT)/SSD**	**Function**	**PLSHADE**	**PSHADE**	**PUSHADE(T)/SSD**	**SHADE-ILS**	**PUSHADE(DPT)/SSD**	**Function**	**PLSHADE**	**PSHADE**	**PUSHADE(T)/SSD**	**SHADE-ILS**	**PUSHADE(DPT)/SSD**	**Function**	**PLSHADE**	**PSHADE**	**PUSHADE(T)/SSD**	**SHADE-ILS**	**PUSHADE(DPT)/SSD**
1	−	+	≈	+	1.6634e+08	1	−	−	−	+	1.5510e-01	1	≈	≈	≈	−	0.0000e+00	1	≈	≈	≈	−	0.0000e+00
2	−	−	≈	+	6.0305e+03	2	+	−	≈	+	4.1529e+03	2	+	−	≈	+	4.1529e+03	2	+	−	≈	+	4.1529e+03
3	−	+	≈	+	2.1324e+01	3	−	+	−	−	2.0030e+01	3	−	+	−	−	2.0000e+01	3	−	≈	−	−	2.0000e+01
4	−	+	≈	+	6.3678e+09	4	−	−	≈	−	1.0196e+08	4	+	−	≈	−	3.6044e+07	4	+	−	≈	−	7.8394e+06
5	−	−	≈	≈	1.8080e+06	5	−	−	−	−	4.9304e+05	5	≈	−	−	−	4.6008e+05	5	+	−	−	−	4.6006e+05
6	≈	+	≈	+	1.0602e+06	6	−	≈	−	+	1.0570e+06	6	−	+	−	+	1.0529e+06	6	−	≈	−	+	1.0452e+06
7	−	+	≈	+	1.4083e+07	7	−	−	≈	+	6.1277e+03	7	−	−	≈	≈	5.3002e-03	7	≈	−	≈	−	0.0000e+00
8	−	≈	≈	−	1.3690e+11	8	−	−	≈	−	1.5229e+09	8	+	−	≈	−	2.6637e+08	8	+	−	≈	−	6.1084e+07
9	−	−	≈	+	3.1699e+08	9	−	−	≈	−	1.3554e+08	9	+	−	≈	≈	1.3553e+08	9	+	−	≈	≈	1.3553e+08
10	≈	≈	≈	+	9.4140e+07	10	≈	+	≈	+	9.3820e+07	10	−	+	≈	+	9.2969e+07	10	≈	≈	≈	+	9.2297e+07
11	−	+	≈	+	5.3015e+08	11	−	−	≈	+	5.6524e+05	11	−	−	+	+	1.6498e+04	11	+	−	+	+	7.4348e+02
12	−	−	≈	+	1.2948e+06	12	−	−	≈	+	1.2392e+03	12	−	+	+	+	4.7504e+02	12	−	+	≈	≈	3.5409e+01
13	−	+	≈	+	6.7059e+08	13	−	−	≈	+	2.6992e+05	13	−	−	≈	+	5.5954e+03	13	+	−	≈	+	5.0833e+02
14	−	≈	≈	+	6.4080e+08	14	−	−	≈	+	6.3172e+06	14	≈	−	≈	+	4.6861e+06	14	+	−	≈	+	4.6279e+06
15	−	+	+	+	8.9915e+07	15	−	+	≈	+	2.7918e+07	15	≈	+	≈	+	1.4831e+07	15	≈	+	≈	+	8.7093e+06

[4] R. Tanabe and A. Fukunaga. Success-history based parameter adaptation for differential evolution. In *2013 IEEE Congress on Evolutionary Computation (CEC)*, pages 71–78. IEEE, 2013.

[5] R. Tanabe and A.S. Fukunaga. Improving the search performance of shade using linear population size reduction. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 1658–1665. IEEE, 2014.

[6] M. Sepesy and J. Brest. A review of the recent use of differential evolution for large-scale global optimization: An analysis of selected algorithms on the CEC 2013 LSGO benchmark suite. *Swarm and Evolutionary Computation*, 50:100428, 2019.

[7] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 4th edition, 2022.

[8] R. Storn and K.V. Price. Minimizing the real functions of the icec'96 contest by differential evolution. In *IEEE International Conference on Evolutionary Computation (ICEC)*, pages 842–844. IEEE, 1996.

[9] R. Tanabe and A. Fukunaga. Reviewing and benchmarking parameter control methods in differential evolution. *IEEE Trans. Cybern.*, 50(3):1170–1184, 2020.

[10] D. Molina, A. LaTorre, and F. Herrera. SHADE with iterative local search for large-scale global optimization. In *Proc. 2018 IEEE Congress on Evolutionary Computation*. IEEE, 2018.

[11] Samsung Electronics. *Samsung NVMe SSD 9100 PRO with Heatsink Datasheet, Rev. 2.0*, December 2025. URL: https://download.semiconductor.samsung.com/resources/data-sheet/Samsung_NVMe_SSD_9100_PRO_with_Heatsink_Datasheet_Rev.2.0.pdf.