Distributed Hyper-Heuristics for Real Parameter Optimization*

Marco Biazzini
University of Trento
Trento, Italy
biazzini@dit.unitn.it

Balázs Bánhelyi
University of Szeged
Szeged, Hungary
banhelyi@
inf.u-szeged.hu

Alberto Montresor
University of Trento
Trento, Italy
montreso@dit.unitn.it

Márk Jelasity
Univ. of Szeged and HAS
Szeged, Hungary
jelasity@
inf.u-szeged.hu

ABSTRACT

Hyper-heuristics (HHs) are heuristics that work with an arbitrary set of search operators or algorithms and combine these algorithms adaptively to achieve a better performance than any of the original heuristics. While HHs lend themselves naturally for distributed deployment, relatively little attention has been paid so far on the design and evaluation of distributed HHs. To our knowledge, our work is the first to present a detailed evaluation and comparison of distributed HHs for real parameter optimization in an island model. Our set of test functions includes well-known benchmark functions and two realistic space-probe trajectory optimization problems. The set of algorithms available to the HHs include several variants of differential evolution, and uniform random search. Our main conclusion is that some of the simplest HHs are surprisingly successful in a distributed environment, and the best HHs we tested provide a robust and stable good performance over a wide range of scenarios and parameters.

Categories and Subject Descriptors

G.1.6 [**Numerical Analysis**]: Optimization—*global optimization*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms, Performance, Reliability, Experimentation

*This work was supported by the European Space Agency through Ariadna Project "Gossip-based strategies in global optimization" (21257/07/NL/CB), and by the project CASCADAS (IST-027807) funded by the FET Program of the European Commission. M. Jelasity was supported by the Bolyai Scholarship of the Hungarian Academy of Sciences. B. Bánhelyi was supported by Aktion Österreich-Ungarn 70öu1, and OTKA T 048377.

GECCO'09, July 8–12, 2009, Montréal Québec, Canada.

Keywords

hyper-heuristics, differential evolution, distributed computing

1. INTRODUCTION

Hyper-heuristics are high level problem independent heuristics that work with any set of problem dependent heuristics and adaptively apply and combine them to solve a specific problem [5, 6, 16].

The difference between HHs and meta-heuristics is that meta-heuristics are not off-the-shelf methods that can be readily applied to any problem: they are schemes that have to be instantiated and tuned to specific problems. In contrast to this, HHs do work off-the-shelf using any given set of operators and algorithms. The tradeoff is that HHs are "good enough, soon enough, cheap enough" [6] approaches while meta-heuristics can achieve better performance although they require significantly more investment.

Although it is a promising and useful idea to design and apply parallel HHs, relatively little work has been done in this area, compared to the significant body of work on parallel meta-heuristics [2]. In [19], a master-slave model is proposed, along with a more distributed model where there are many clusters that implement a master-slave model locally. In [15] an agent based approach is proposed that is nevertheless also conceptually centralized involving a single HH agent. Finally, in [22] a Grid-based solution is proposed with a central HH server and slave nodes performing low-level search.

We believe that emerging platforms such as cloud computing [9], as well as the more established peer-to-peer [3] and Grid [14] platforms all favor a coarse grained, decentralized approach that has no bottlenecks and that scales well and tolerates failure and dynamism. Our goal is to target such platforms.

In this paper we examine a set of distributed HHs that are based on an island model, where islands communicate through various scalable and fault tolerant gossip protocols [8]. We compare these HHs empirically over a set of real parameter optimization problems, including realistic space-trajectory optimization problems. Our conclusion is that distributed HHs are competitive optimizers (for example, we could improve the best known solution for one of the realistic problems in our test set), but—most importantly—HHs are robust and consistently better than any of the basic heuristics they apply over a wide range of environments.

code	name
A1	DE/best/1/exp
A2	DE/localBest/1/exp
A3	DE/rand/1/exp
A4	DE/rand/2/exp
A5	DE/randToBest/1/exp
A6	DE/randToLocalBest/1/exp
A7	particle swarm optimization
A8	random sample

Table 1: The set of heuristics $\mathcal{A}$ input to the HHs

2. THE ISLAND MODEL

Our parallelization approach is based on a symmetric island model: we assume that we are given independent nodes, each of which runs the same algorithm, periodically communicating with each other. From now on we use the words "node" and "island" interchangeably.

The neighborhood structure is random. More precisely, we assume that at any point in time each node can request a random node address from a local *peer sampling service* that returns a random sample taken from the network.

While we do not focus on system-level implementation details of the parallel algorithms, we note that the peer sampling service can be implemented in a robust, cheap, and flexible way that scales to millions of nodes [13]. From this point we simply assume that this service is accessible at every node. All the communication mechanisms we will define are based on gossip algorithms [8] that can be implemented on top of this service alone. An actual implementation of a similar framework is available as well [3].

Note that in this framework it would also be possible to use gossip algorithms [11] to generate better neighborhood structures [7, 21]. For simplicity, in this paper we opted for a random structure.

Independently of the algorithm run on the island, we always propagate the current best solution to all the islands. This is also done through a gossip protocol: islands periodically send the best solution they know of to other random islands, and when they receive such a message, they update their own current best solution. We assume the period of gossip to be one function evaluation, which presupposes that the function is non-trivial and takes a sufficiently long time (in the order of a second or more) to compute. It can be shown that the time to propagate a new current best solution to every node this way takes $O(\log N)$ periods in expectation where N is the network size [18].

3. ALGORITHMS

3.1 The Basic Heuristics

Here we describe the set of algorithms $\mathcal{A}$ that our HHs will operate on. A typical HH takes low level operators often classified as simple hillclimbers and mutation operators [16]. Instead, in our approach the HH operates over meta-heuristics as well. These meta-heuristics can still be classified as leaning towards exploration (diversification) or towards exploitation (intensification); the presence of both kinds of algorithms is crucial for every HH.

The set of algorithms is shown in Table 1. Heuristics A1-A6 are variants of differential evolution. We use the standard notation as proposed in [20]. Here, "best" means the global best solution in the network (as learned through gossip, see above). Notation "localBest" in A2 implies the "best" variant with the best solution interpreted as the local best solution within the island: this variant ignores the global best solution so the islands are isolated. Similarly, "rand" variants are also defined to be local to the island. Heuristic A5 is like the "2" variants but it uses one random solution from the population along with the global best; A6 is the isolated version of A5.

Algorithm A7 is described in [4]. It is a simple island-based PSO algorithm that assumes that the best solution PSO relies on is the global best, propagated via gossip. Finally, A8 returns a random solution from the range of the function at hand.

All these algorithms are population-based (except A8, which is stateless). We assume that all the islands maintain a population of size 8. This makes it possible for a HH to change the algorithm while preserving the population.

3.2 Baseline HHs

We include in our pool two trivial HHs as a baseline. The first is called StatEq that is short for "static equal share". StatEq assigns a heuristic to each island at the beginning of the run and does not change this assignment anymore. Furthermore, it assigns an equal number of islands to every heuristic. Note that StatEq can easily be implemented as a local algorithm without global consensus, if necessary: for example, each node can select an algorithm at random at the beginning, and then adhere to it throughout the run (depending on network size, this introduces some variance).

The second is called DynEq that is short for "dynamic equal share". It assigns a random heuristic to each island after each cycle (where one cycle within an island represents the generation of one new solution using a heuristic) at random, giving an equal probability to all the heuristics.

3.3 Tabu

Our first non-trivial HH is adapted from related work [6]. Like the set of heuristics A1-A8, and all the rest of the HHs, this algorithm is run on every island.

In the original sequential version, the basic idea was that Tabu maintains a tabu list of heuristics, and it also maintains a rank value for every heuristic, that can take an integer value from the interval $[0, |\mathcal{A}|]$. If running a heuristic improved the current best solution, its rank is increased by one and the tabu list is emptied; otherwise its rank is decreased by one and it is put in the tabu list. In each cycle, Tabu selects the heuristic that has a highest rank among those that are not in the tabu list.

Note that the tabu list has a dynamic size because it becomes empty whenever an algorithm can improve the current best solution [6]. Its maximal size is $|\mathcal{A}| - 1$.

We parallelize this algorithm by running it on all the islands, but using the global current best solution for the improvement test (recall that the current best solution is known locally via gossip). In addition, when we learn about a new global current best solution from a neighbor (along with the heuristic that generated it), we treat this event exactly as if the improvement was the result of running the given heuristic locally (that is, we update the rank of the given heuristic, and so on).

3.4 SDigmo and DDigmo

A parallel master-slave HH called Digmo was proposed in [22] designed for a local Grid environment within the European Space Agency. Here we propose two fully distributed adaptations of this method for our island model.

The basic idea behind Digmo is that it maintains a probability distribution over the algorithm set $\mathcal{A}$ based on the performance of the algorithms. It uses a master-slave architecture, where the master keeps a central population, and periodically selects algorithms based on the probability distribution; it then assigns the selected algorithm to a slave node along with a random subset of the central population. When an algorithm reports the results back to the master, the master updates the probability distribution and the central population as well.

For all the algorithms, Digmo maintains a FIFO queue of size k, that contains the k last results of the algorithm (they recommend a value of $k = 5$). Let M_i denote the average of these k values for algorithm i. In the case of minimization and a positive function, the probabilities P_i are chosen so as to be proportional to $1/M_i$:

$$P_i = \alpha \frac{1}{sM_i} + (1-\alpha)\frac{1}{|\mathcal{A}|}, \qquad s = \sum_{j=1}^{|\mathcal{A}|} \frac{1}{M_i},$$

where $0 < \alpha < 1$ is a constant that determines the minimal probability each algorithm is assigned. A setting of $\alpha = 0.2$ is suggested.

We adapt this algorithm to our island model by allowing each island to approximate P_i for all i, and then allowing the islands to cooperatively assign heuristics for each island in two different (static and dynamic) ways based on this distribution.

To approximate P_i, each island maintains a good approximation of the FIFO queue for each heuristic, via gossiping the latest results of the algorithms. Thus, the queue of algorithm i will contain the last k results of i in the entire network, with a small time lag due to gossip propagation delay. This way, M_i, and thus P_i, can be approximated locally at each island.

Knowing P_i for all i, the *dynamic* way of assigning heuristics is simply to pick a heuristic at random using this distribution independently at every island in every cycle. We call this variant DDigmo.

In the static approach—that we call SDigmo—we still want the network to reflect the distribution P_i, however, we want minimize the number of islands that actually change their heuristic during a run. For this an island needs extra information: an approximation of the actual proportion of algorithm i in the network, denoted by $\hat{P}_i$. An island running algorithm i will keep running i if $P_i \geq \hat{P}_i$. Otherwise it will select a novel algorithm j with a probability proportional to $\max\{0, P_j - \hat{P}_j\}$.

For the local approximation of $\hat{P}_i$ we apply gossip-based aggregation [12]. This protocol has an identical cost and time complexity to gossip based multicast that we apply for propagating the global best solution, and it also assumes only the peer sampling service to be able to function properly. The basic idea behind it is simulating diffusion and thereby calculating averages, network size, and other statistics.

In SDigmo and DDigmo, based on extensive preliminary experiments, we set $\alpha = 1$ and $k = 5$.

Algorithm 1 Pruner HH

```
1: for r ← 1 to I do
2:     n_e ← ⌈|A|(I − r)/I⌉
3:     if n_e has changed or newBest then
4:         newBest ← false
5:         rank ← sort(stats)
6:         i ← lookup(rank, curr)
7:         if i > n_e then
8:             i ← 1
9:         else
10:            i ← max(0, i − 1)
11:        end if
12:        curr ← rank[i]
13:    end if
14:    val ← run(curr, bestVal)
15:    updateStats(val, curr)
16:    p ← getRandomPeer()          ▷ peer sampling service
17:    send ⟨bestVal, bestAlg⟩ to p
18: end for
19: procedure updateStats(val, alg)
20:    if val is better than bestVal then
21:        bestVal ← val
22:        bestAlg ← alg
23:        stats[alg] ← val
24:    end if
25: end procedure
26: procedure onReceive(⟨val, alg⟩)
27:    if val is better than bestVal then
28:        newBest ← true
29:    end if
30:    updateStats(val, alg)
31: end procedure
```

3.5 Pruner

The main motivation for applying HHs is arguably their ability to adaptively combine search diversification and intensification in order to produce good solutions. However, in our case, since we apply meta-heuristics as a set of basic heuristics, it might also make sense to try and pick the one that best fits the problem at hand, since meta-heuristics themselves could deal with balancing between exploration and exploitation to a certain degree, with varying success depending on the problem.

The Pruner HH is designed with this idea in mind. It initially uses the entire collection of available algorithms $\mathcal{A}$, but as the search proceeds, it removes more and more algorithms from this set and does not consider them anymore. At any given time, we will call the set of algorithms that are still being considered the *eligible* set.

We decrease the size of the eligible set according to a schedule that is defined by the maximal number of iterations (or cycles) I that is assigned to each island. Recall that in each cycle we evaluate one new solution. The size of the eligible set in cycle r is $|\mathcal{A}|(I - r)/I$.

The main idea here is that a node applies the same algorithm until either the number of eligible algorithms decreases, or a new current best solution is received from another node via gossip. When any of these events occur, Pruner sorts the algorithms according to the best results they have produced so far and attempts to choose an algorithm that is better than the current one.

The Pruner HH is given in Algorithm 1. In this algorithm, *stats* stores, for each heuristic, the best solution found so far. Array *rank* is a sorted list of the algorithms (from best to worst) based on the information contained in *stats*. Variable *curr* holds the current algorithm.

In each cycle Pruner first computes the number n_e of *eligible* algorithms. If n_e has changed from the previous iteration, or a recent gossip message has updated the best known solution, the current algorithm *curr* to be used for subsequent run is updated as follows. First, the position of algorithm *curr* in the sorted list of algorithms *rank* is obtained through the lookup call. If the current algorithm is no longer eligible, we switch to the best algorithm available (that is, *rank*[1]). Otherwise, the algorithm one rank better than the current algorithm is chosen.

If none of the events occur, then nothing happens: the current algorithm is not changed.

It is important to note that—since each node manages its own eligible set that can differ—Pruner can occasionally add a removed algorithm again if a result is received via gossip that ranks the given algorithm high enough.

3.6 Scanner

Apart from shrinking the eligible set in the same way as Pruner, the key idea behind Scanner is to provide an opportunity for each algorithm in order to get a better picture of the performance of a given algorithm, and also to allow for possible synergic effects among the algorithms.

To achieve this, we introduce two notions. First, we define a minimal number of consecutive executions for each heuristic (building on the fact that our heuristics can themselves jump out of local optima). Second, we keep iterating over all the algorithms in the current eligible set and give all of them the minimal number of consecutive executions (scanning).

Scanner is listed in Algorithm 2. Here, *stats*[*a*] stores the *latest* solution obtained by algorithm *a*. Additional variables are *rank*, a sorted list based on *stats*; *counter*, the number of non-improving iterations for the current algorithm; and *phase*, a state variable that stores the current phase of the algorithm: SCAN or NORMAL. Function MaxNonImproving(*phase*) takes the phase as input and returns the maximum number of consecutive non-improving iterations any algorithm is allowed to take.

Scanner is organized in two distinct phases. Phase SCAN is activated whenever a gossip message containing a new best solution is received. At that point, algorithms are sorted based on the latest solutions they have found so far (stored in *stats*) and variables are initialized in order to start scanning from the first algorithm. Subsequently, a few iterations for each of the eligible algorithms are executed, having the goal of verifying whether the new solution just received can be further improved by the remaining eligible algorithms.

When all the eligible algorithms have been tested, we switch to phase NORMAL. In this phase we keep scanning the same way as in phase SCAN except that the maximal number of non-improving iterations is larger and depends on time as well.

The exact formula we use is MaxNonImproving(NORMAL) $= \lceil I/(c \cdot n_e) \rceil$, and MaxNonImproving(SCAN) $=$ min(15, MaxNonImproving(NORMAL)/2), where n_e is the size of the eligible set and c is the number of iterations during which the current algorithm has been kept to be the current algorithm continuously. Note that since n_e can change, this recursive formula cannot be solved exactly independently of time, but nevertheless it is approximately $\sqrt{I/n_e}$. This setting, as well as all other design decisions, are a result of extensive preliminary experiments with earlier versions and alternatives.

Algorithm 2 Scanner HH

```
1: for r ← 1 to I do
2:     if newBest then
3:         newBest ← false
4:         rank ← sort(stats)
5:         i ← 1
6:         counter ← 0
7:         phase ← SCAN
8:     end if
9:     val ← run(rank[i], bestVal)
10:    counter = UPDATESTATS(val, rank[i])
11:    if counter > MaxNonImproving(phase) then
12:        counter ← 0
13:        i ← i + 1
14:    end if
15:    if i = ⌈|A| · (I − r)/I⌉ then    ▷ Eligible group size
16:        if phase = SCAN then
17:            rank ← sort(stats)
18:            phase ← NORMAL
19:        end if
20:        i ← 1
21:    end if
22:    p ← getRandomPeer()        ▷ peer sampling service
23:    send ⟨bestVal, bestAlg⟩ to p
24: end for
25: procedure UPDATESTATS(val, alg)
26:    stats[alg] ← val
27:    if val is better than bestVal then
28:        bestVal ← val
29:        bestAlg ← alg
30:        return 0
31:    else
32:        return counter + 1
33:    end if
34: end procedure
35: procedure ONRECEIVE(⟨val, alg⟩)
36:    if val is better than bestVal then
37:        newBest ← true
38:    end if
39:    UPDATESTATS(val, alg)
40: end procedure
```

4. EXPERIMENTAL RESULTS

The experiments were run using PeerSim, a network simulator originally developed for experimenting with large scale peer-to-peer protocols, such as gossip-based multicast and aggregation [17]. The source code of the PeerSim implementation of the HHs is available from the PeerSim homepage.

In the following we outline the experimental setup and then discuss the results obtained.

4.1 Test Functions

We chose well-known test functions as shown in Table 3. We included Sphere10 as an easy unimodal function. Rosenbrock10 and Zakharov10 are included as non-trivial uni-

name	short description
StatEq	equal share for heuristics in space
DynEq	equal share for heuristics in time
Tabu	an island based version of [6]
SDigmo	static variant of the HH inspired by [22]
DDigmo	dynamic variant of the HH inspired by [22]
Pruner	focusing search on best heuristics
Scanner	attempting to give a chance to every heuristic

Table 2: Summary of our pool of HHs

modal functions. The rest of the functions are multimodal. Griewank10 is similar to Sphere10 with high frequency sinusoidal "bumps" superimposed on it. Schaffer10 is a sphere-symmetric function where the global minimum is surrounded by deceptive spheres. Levy4 is not unlike Griewank10, but is more asymmetric, and involves higher amplitude noise too.

Cassini1 and Cassini2 are realistic applications related to the Cassini spacecraft trajectory design problem of the European Space Agency (ESA). The two problems have 6 and 22 real variables, respectively, and an unknown number of local optima. These problems have been extensively studied and are known to contain an enormous number of local optima and to be strongly deceptive for local optimizers [1].

4.2 Experimental Setup

In our experiments we varied the following parameters:

- **network size** (N) the number of nodes (islands) in the network
- **function evaluations** (E) the number of *overall* function evaluations performed in the network

For a combination of network size N and overall function evaluations E, each island is assigned an equal number of function evaluations: E/N.

We ran 10 independent experiments for each combination of E and N where

$$N \in \{2^0, 2^1, \ldots, 2^{16}\} \text{ and } E \in \{2^{10}, 2^{13}, 2^{17}, 2^{20}\},$$

for every possible algorithm in Table 2 *and* the standalone versions of the algorithms in Table 1, on each test function.

The outcome of a single experiment is the best solution found in the network.

4.3 Filtering the Raw Outcome

Our primary goal is to compare the algorithms from the point of view of stability and reliable good performance across a wide range of parameters, since these are the trademark features of a good HH.

To clean the generated data from noise, before analyzing the results we first selected only one value for parameter E for each function. The reason is that if E is too large, then the results are inconclusive: all the algorithms produce almost identical results very close to the global optimum, which makes it impossible to differentiate between the algorithms. This was problematic especially for the very easy functions: Sphere and Zakharov.

If E is very small, then none of the algorithms produce very good results, so comparison is again not really worthwhile. We selected the value that differentiates most among the algorithms: $E = 2^{20}$ for Cassini1, Cassini2, Griewank, Schaffer and Rosenbrock; $E = 2^{17}$ for Levy, and $E = 2^{13}$ for Sphere and Zakharov.

Network size is also important to consider. Large networks ($N \geq 2^{14}$) allow too few evaluations per island even for $E = 2^{20}$, the largest value of E; while in small networks ($N \leq 2^2$) the behavior of the algorithms is rather different than in larger networks, and, quite interestingly, results for the same value of E are of lower quality than in larger networks. Since we are interested in relatively large networks where all the islands still have a reasonable number of evaluations, we removed the experiments with the indicated extremal network sizes.

	Number of times best, 2nd best,. . .,10th best									
StatEq	4	12	8	7	4	7	3	5	3	1
SDigmo	6	4	6	11	10	6	3	1	2	4
Pruner	5	6	11	7	7	4	3	3	3	3
A1	9	2	1	3	5	11	7	3	2	2
Scanner	7	5	6	1	5	2	5	2		5
A4	8	7	1	1	3	4	3	5	3	5
DynEq	2	2	3	4	2	5	7	7	7	5
DDigmo	1	3	4	3	2	2	9	8	6	7
A5	4	3	4	5	4	2	1	3	1	4
A7	5	6	2	1		3	2	7	3	3
A6	2	1	2	7	4	2	3	3	2	1
A3	2	2	4	3	1	1	3	1	5	8
Tabu	1	2	2		3	3	4	5	8	4
A2		1	2	2	4		1	2	7	2
A8				1	2	4	2	1	4	2

Table 5: Mean best fitness rank statistics.

4.4 Dominance Analysis

In the remaining data, we were interested in characterizing dominant protocols, that perform well in every case. To achieve this, we calculated the dominance matrix as shown in Table 4. In this matrix, an entry $a_{i,j}$ denotes the number of different parameter settings, where the *average* of the best value found during each of the 10 independent runs (also called the *mean best fitness* measure) by algorithm i (column index) was better than that of algorithm j (row index). The sum $a_{i,j} + a_{j,i}$ is the number of different parameter settings, that is, the number of different types of experiments in the dataset.

In addition, we also list ranking information for the mean best fitness in Table 5. In the table the first column contains the number of different parameter settings where the mean best fitness of the given algorithm was best; the second column contains the number of times it was second best, and so on.

These two tables together offer interesting insights into the performance of the algorithms. First of all, we can see that the most dominant HH is one of our baseline heuristics, StatEq. The second best heuristic, SDigmo, is dominated by StatEq by a substantial margin: 34 to 22.

As a general pattern, we see that HHs that tend to be static and do not change the heuristic on an island too often tend to be better (more dominant) than the dynamic variants, so this feature seems to be desirable in an island model.

Another observation is that HHs consistently and very convincingly dominate all the algorithms in $\mathcal{A}$, which clearly underlines the main advantage of HHs. The best performing algorithm according to this measure is A1, which ranks 4th.

Looking at Table 5, however, we notice that A1 has the largest number of wins among the possible parameter set-

	Function $f(x)$	D	$f(x^*)$	K
Sphere10	$\sum_{i=1}^{10} x_i^2$	$[-5.12, 5.12]^{10}$	0	1
Rosenbrock10	$\sum_{i=1}^{9} 100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2$	$[-100, 100]^{10}$	0	1
Zakharov10	$\sum_{i=1}^{10} x_i^2 + (\sum_{i=1}^{10} ix_i/2)^2 + (\sum_{i=1}^{10} ix_i/2)^4$	$[-5, 10]^{10}$	0	1
Griewank10	$\sum_{i=1}^{10} x_i^2/4000 - \prod_{i=1}^{10} \cos\left(x_i/\sqrt{i}\right) + 1$	$[-600, 600]^{10}$	0	$\approx 10^{19}$
Schaffer10	$0.5 + (\sin^2(\sqrt{\sum_{i=1}^{10} x_i^2}) - 0.5)/$ $(1 + (\sum_{i=1}^{10} x_1^2)/1000)^2$	$[-100, 100]^{10}$	0	≈ 63 spheres
Levy4	$\sin^2(3\pi x_1) + \sum_{i=1}^{3}(x_i - 1)^2(1 + \sin^2(3\pi x_{i+1}))+$ $(x_4 - 1)(1 + \sin^2(2\pi x_4))$	$[-10, 10]^4$	-21.502356	71000
Cassini1	description available from ESA at `http://www.esa.int/gsp/ACT/inf/op/globopt/evvejs.htm`			
Cassini2	description available from ESA at `http://www.esa.int/gsp/ACT/inf/op/globopt/edvdvdedjds.htm`			

Table 3: Test functions. D: search space; $f(x^*)$: global minimum value; K : number of local minima.

	StatEq	SDigmo	Pruner	A1	Scanner	A4	DynEq	DDigmo	A5	A7	A6	A3	Tabu	A2	A8	sum
StatEq		34	31	32	35	38	45	47	44	33	46	47	48	47	49	576
SDigmo	22		30	31	35	37	43	46	41	34	42	49	49	49	47	555
Pruner	25	26		38	38	31	43	43	37	39	39	47	43	52	49	550
A1	24	25	18		25	31	39	39	31	41	33	36	36	42	43	463
Scanner	21	21	18	31		29	29	32	32	35	33	40	33	46	44	444
A4	18	19	25	25	27		24	24	39	36	38	34	44	36	41	430
DynEq	11	13	13	17	27	32		20	35	30	38	40	38	45	49	408
DDigmo	9	10	13	17	24	32	36		34	27	35	40	36	45	49	407
A5	12	15	19	25	24	17	21	22		29	38	28	38	34	44	366
A7	23	22	17	15	21	20	26	29	27		28	30	25	33	35	351
A6	10	14	17	23	23	18	18	21	18	28		28	34	32	46	330
A3	9	7	9	20	16	22	16	16	28	26	28		28	40	46	311
Tabu	8	7	13	20	23	12	18	20	18	31	22	28		32	37	289
A2	9	7	4	14	10	20	11	11	22	23	24	16	24		43	238
A8	7	9	7	13	12	15	7	7	12	21	10	10	19	13		162

Table 4: Dominance matrix based on mean best fitness.

	Number of times best, 2nd best,...,10th best									
A1	20		4	2	6	6	5	5		1
StatEq	5	9	6	7	4	8	5	4	2	2
SDigmo	4	7	10	7	7	3	5	3	2	1
Pruner	3	8	6	5	1	4	4	4	4	4
DynEq	2	6		6	4	6	5	5	8	7
A4	4	7	1	5	4	5	5	4	2	7
Scanner		4	8	6	11	5	1	2		
A7	4	4	5	1	3	6	6	9	6	1
DDigmo	5		3	3	4	4	8	7	5	5
Tabu	1	2	1	6	6	4	4	1	4	2
A5	1	3	3	2	1	1	3	5	4	5
A3	2	2	1	2	2	2		2	7	6
A2	5	2	3				1	2	7	5
A6		1	3	3	1	1	3	3	3	5
A8		1	2	1	2	1	1		2	5

Table 6: Minimal best fitness rank statistics.

tings. There is a catch though: its ranking distribution is bimodal: it has another peak at around rank 6; this means that A1 is often the best, but when it is not best, it is rather bad. HHs show a more reliable and stable pattern.

This is even better illustrated by Table 6 which, instead of the mean best fitness, is calculated based on the *best* result of the 10 independent runs: we see that A1 can be very good, but its performance is quite unreliable. The corresponding dominance matrix is shown in Table 7, where the best HHs have the same order, but A1 leaps ahead in dominance.

Naturally, dominance depends on the set of test functions we have examined. We tried to remove the easiest functions from the dataset: Sphere and Zakharov. These functions are too easy for most of the algorithms so they should be given less weight in the comparison. On the dataset without the easy functions, we see a slightly altered dominance matrix (Table 8). Algorithm A1 now seems less favorable: it turns out A1 excels on the easy functions primarily. However, the best three HHs are still the same as in Table 4, which provides further evidence that a good HH can in fact achieve a better performance than any of the basic algorithms it is based on, and that this performance is rather stable as well.

4.5 Statistical Tests

Before turning to a more fine-grained presentation of the performance of the algorithms, we first discuss whether the algorithms that have a similar dominance pattern are in fact significantly different. Recall that we have a sample of size 10 for each parameter setting. For a pair of algorithms i and j we can ask ourselves whether their samples are significantly different in a statistical sense?

Since we have no information about the underlying distribution, and we have no reason to assume that it is Gaussian, we use a nonparametric statistical test, the Mann-Whitney test [10], to decide whether we can significantly differentiate between i and j based on the 10 samples. The results are somewhat surprising: the difference between StatEq and SDigmo is not statistically significant (at level 5%) in the vast majority of parameter settings. The difference between DDigmo and DynEq is not significant either. This is consistent with the similar rank of these pairs in Tables 4 and 8.

For the rest of the algorithm pairs we could not find any other clear case where the difference could be questioned.

	A1	StatEq	SDigmo	Pruner	DynEq	A4	Scanner	A7	DDigmo	Tabu	A5	A3	A2	A6	A8	sum
A1		36	31	36	42	40	35	42	42	40	42	43	41	43	45	558
StatEq	20		41	40	40	30	36	29	45	46	43	44	42	44	48	548
SDigmo	25	15		40	44	33	43	29	46	46	42	46	44	43	46	542
Pruner	20	16	16		25	29	41	27	28	43	40	40	39	42	45	451
DynEq	14	16	12	31		34	29	24	21	37	42	46	45	44	47	442
A4	16	26	23	27	22		25	33	27	36	45	38	34	43	46	441
Scanner	21	20	13	15	27	31		31	31	36	41	43	43	40	45	437
A7	14	27	27	29	32	23	25		37	30	35	38	39	37	42	435
DDigmo	14	11	10	28	35	29	25	19		34	37	45	45	41	42	415
Tabu	16	10	10	13	19	20	20	26	22		35	34	34	37	38	334
A5	14	13	14	16	14	11	15	21	19	21		27	28	36	43	292
A3	13	12	10	16	10	18	13	18	11	22	29		34	32	41	279
A2	15	14	12	17	11	22	13	17	11	22	28	22		28	46	278
A6	13	12	13	14	12	13	16	19	15	19	20	24	28		42	260
A8	11	8	10	11	9	10	11	14	14	18	13	15	10	14		168

Table 7: Dominance matrix based on minimal best fitness.

	StatEq	SDigmo	Pruner	A4	A5	A1	A6	Scanner	DDigmo	DynEq	Tabu	A7	A3	A2	A8	sum
StatEq		27	27	26	31	26	33	33	37	37	36	28	35	35	35	446
SDigmo	15		26	24	27	25	28	30	36	33	35	28	35	35	33	410
Pruner	15	16		19	23	31	25	31	32	32	30	32	34	39	35	394
A4	16	18	23		25	22	25	25	22	22	30	32	30	32	30	352
A5	11	15	19	17		24	28	23	21	20	27	26	28	34	36	329
A1	16	17	11	20	18		20	18	29	30	23	32	25	31	29	319
A6	9	14	17	17	14	22		22	20	17	24	26	28	32	38	300
Scanner	9	12	11	17	19	24	20		20	19	21	28	28	34	30	292
DDigmo	5	6	10	20	21	13	22	22		27	24	21	28	33	35	287
DynEq	5	9	10	20	22	12	25	23	15		25	22	28	33	35	284
Tabu	6	7	12	12	15	19	18	21	18	17		28	28	32	31	264
A7	14	14	10	10	16	10	16	14	21	20	14		20	23	21	223
A3	7	7	8	12	14	17	14	14	14	14	14	22		34	32	223
A2	7	7	3	10	8	11	10	8	9	9	10	19	8		29	148
A8	7	9	7	12	6	13	4	12	7	7	11	21	10	13		139

Table 8: Dominance matrix based on mean best fitness, excluding Sphere and Zakharov from the dataset.

4.6 Performance on Test Functions

Based on the Mann-Whitney tests, and the fact that StatEq dominates SDigmo, we exclude SDigmo from further consideration. Taking this into account, and based on the dominance results, we identify StatEq, Pruner, and Scanner as the best HHs, and A1 and A4 as the best basic heuristics. Figure 1 illustrates mean best fitness as a function of network size for the non-trivial test functions.

We notice that StatEq is very stable and tends to be at the lower bound of the other algorithms (or it is the best, see Cassini1) except for a few special cases where A4 and Scanner perform well.

Finally, we note that Scanner actually improved the best known solution to Cassini1.[1] Scanner, Pruner and SDigmo produced competitive results for Cassini2 as well, e.g. SDigmo reached 8.410157744690402, although with tuned parameters and $E = 2^{23}$. However, this might serve as an reminder that although StatEq is the most stable dominant method, and as such the most preferable HH in our set, for specific problems other heuristics might yield a better peak performance.

5. CONCLUSIONS

In this paper we provided convincing evidence through an extensive experimental analysis that a conceptually very simple baseline method is a quite competitive HH in a *large scale parallel environment* using a standard island model.

We also presented promising HHs such as Pruner and Scanner that show a competitive performance with respect to both dominance and peak performance as well on certain problems.

It is also clear that this environment favors conservative methods, that is, an island should not change its heuristic very often. This could be due to the fact that variants of differential evolution, that we mainly use as basic heuristics due to their competitive performance and simple configuration, strongly depend on the population distribution.

As a last note, we point out that although in our experiments SDigmo did not turn out to be statistically different from StatEq, it outperformed both Scanner and Pruner, and, in general, we consider it a promising algorithm. More research is needed to find out whether there are problems or parameter settings where SDigmo may actually be significantly superior to StatEq.

6. REFERENCES

[1] B. Addis, A. Cassioli, M. Locatelli, and F. Schoen. Global optimization for the design of space trajectories, 2008. Optimization Online eprint archive `http://www.optimization-online.org/DB_HTML/2008/11/2150.html`.

[2] E. Alba. *Parallel Metaheuristics: A New Class of Algorithms*. Wiley, 2005.

[3] M. G. Arenas, P. Collet, A. E. Eiben, M. Jelasity, J. J. Merelo, B. Paechter, M. Preuß, and M. Schoenauer. A framework for distributed evolutionary algorithms. In

[1] Cassini1(−789.7652528252638, 158.30958439573675, 449.38588149034445, 54.713196036801925, 1024.7266958960276, 4552.859162641155) = 4.930707804754513

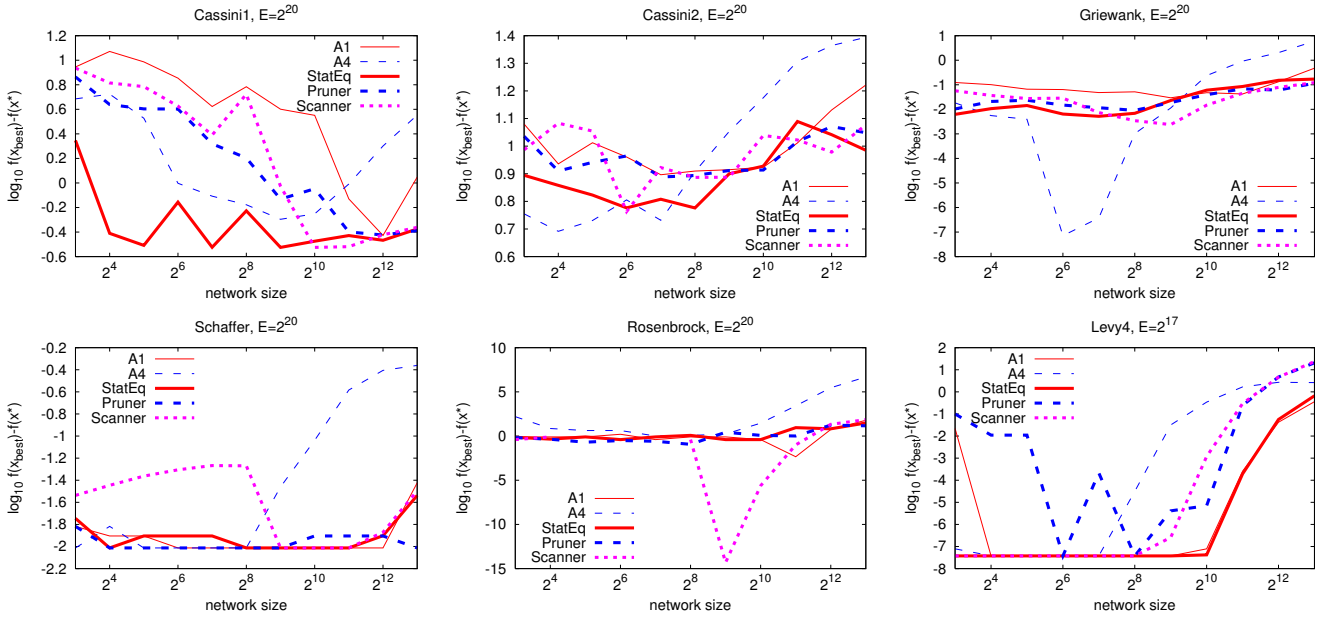

Figure 1: Mean best fitness (expressed as the difference from the optimal solution) on the non-trivial test functions as a function of network size.

J. J. Merelo Guervós, P. Adamidis, H.-G. Beyer, J.-L. Fernández-Villacañas, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VII*, volume 2439 of *LNCS*, pages 665–675. Springer, 2002.

[4] M. Biazzini, A. Montresor, and M. Brunato. Towards a decentralized architecture for optimization. In *Proc. of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, Miami, FL, USA, April 2008.

[5] E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In *Handbook of Metaheuristics*, pages 457–474. 2003.

[6] E. K. Burke, G. Kendall, and E. Soubeiga. A tabu-search hyperheuristic for timetabling and rostering. *Journal of Heuristics*, 9(6):451–470, 2003.

[7] E. Cantu-Paz. *Efficient and Accurate Parallel Genetic Algorithms*. Springer, 2000.

[8] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12. ACM Press, August 1987.

[9] E. Hand. Head in the clouds. *Nature*, 449:963, 2007.

[10] M. Hollander and D. A. Wolfe. *Nonparametric Statistical Methods*. Wiley, 2nd edition, 1999.

[11] M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay topology management. In S. A. Brueckner, G. Di Marzo Serugendo, D. Hales, and F. Zambonelli, editors, *Engineering Self-Organising Systems*, volume 3910 of *LNCS*, pages 1–15. Springer, 2006.

[12] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems*, 23(3):219–252, August 2005.

[13] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3):8, August 2007.

[14] J. Joseph and C. Fellenstein. *Grid Computing*. Prentice Hall, 2003.

[15] D. Ouelhadj and S. Petrovic. A cooperative distributed hyper-heuristic framework for scheduling. In *Proc. of the IEEE Int conference on Systems, Man and Cybernetics (SMC 2008)*, Singapore, 2008.

[16] E. Özcan, B. Bilgin, and E. E. Korkmaz. A comprehensive analysis of hyper-heuristics. *Intelligent Data Analysis*, 12(1):3–23, 2008.

[17] PeerSim. http://peersim.sourceforge.net/.

[18] B. Pittel. On spreading a rumor. *SIAM Journal on Applied Mathematics*, 47(1):213–223, February 1987.

[19] P. Rattadilok, A. Gaw, and R. S. Kwan. Distributed choice function hyper-heuristics for timetabling and scheduling. In *Practice and Theory of Automated Timetabling PATAT V*, number 3616 in LNCS, pages 51–67. Springer, 2005.

[20] R. Storn and K. Price. Differential evolution: A simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, December 1997.

[21] M. Tomassini. *Spatially Structured Evolutionary Algorithms: Artificial Evolution in Space and Time*. Springer, 2005.

[22] T. Vinkó and D. Izzo. Learning the best combination of solvers in a distributed global optimization environment. In *Proceedings of AGO 2007*, pages 13–17, Mykonos, Greece, 2007.