Gossip-based Strategies in Global Optimization

Final Report

Authors: Mark Jelasity, Balazs Banhelyi
Affiliation: University of Szeged, Hungary

Authors: Marco Biazzini, Alberto Montresor
Affiliation: University of Trento, Italy

ESA Researcher(s): Dario Izzo, Tamas Vinko

Date: 2009/01/29, Szeged, Hungary

Contacts:

Mark Jelasity
Tel: +36 62 544127
Fax: +36 62 425508
e-mail: jelasity@inf.u-szeged.hu
Leopold Summerer
Tel: +31(0)715655174
Fax: +31(0)715658018
e-mail: act@esa.int

Available on the ACT website
http://www.esa.int/act

Ariadna ID: 07/5201
Study Duration: 4 months
Contract Number: 21257/07/NL/CB

Contents

Chapter 1

Introduction

The work we have completed was organized around two topics. The first was the investigation of the possibilities for implementing various global optimization approaches in a peer-to-peer environment, using gossip algorithms. We compared two rather different algorithms: a branch-and-bound method and particle swarm optimization. The results are presented in Chapter 2.

The second topic of investigation was the fully distributed implementation of hyper-heuristics, that manage a pool of lower level algorithms and try to adaptively combine or filter them for solving a given problem. The local Grid of ESA already applies such a hyper-heuristic called Digmo [36]. We conducted an extensive study during which we proposed various fully distributed versions of Digmo along with a number of original algorithms. The results of this study is presented in Chapter 3.

Chapter 2

Peer-to-peer Optimization with Branch-and-Bound and Particle Swarms

Recent developments in the area of peer-to-peer (P2P) computing have enabled a new generation of fully-distributed global optimization algorithms via providing self-organizing control and load balancing mechanisms in very large scale, unreliable networks. Such decentralized networks (lacking a GRID-style resource management and scheduling infrastructure) are an increasingly important platform to exploit. So far, little is known about the scaling and reliability of optimization algorithms in P2P environments. In this chapter we present empirical results comparing two algorithms for real-valued search spaces in large-scale and unreliable networks. The two algorithms that are compared are a known distributed particle swarm optimization (PSO) algorithm and a novel P2P branch-and-bound (B&B) algorithm based on interval arithmetic. Comparing two rather different paradigms for solving the same problem gives a better characterization of the limits and possibilities of optimization in P2P networks.

2.1 Introduction

During the past decade various large scale networks have emerged as computing platforms such as the Internet, the web, in-house clusters of cheap computers, and, more recently, networks of mobile devices. The exploitation of these networks for computing and for other purposes such as file sharing and content distribution has followed a different path. Whereas computing is normally performed using GRID technologies, other applications, due to legal and efficiency reasons, favored fully decentralized self-organizing approaches, that became known as peer-to-peer (P2P) computing.

It is an emerging area of research to transport P2P algorithms back into the world of scientific computing, in particular, distributed global optimization. P2P algorithms can replace some of the centralized mechanisms of GRIDs that include monitoring and control functions. For example, network nodes can distribute information via "gossiping" with each other and they can collectively compute aggregates of distributed data (average, variance, count, etc) to be

used to guide the search process [23]. This in turn increases robustness and communication efficiency, allows for a more fine-grained control over the parallel optimization process, and makes it possible to utilize large-scale resources without a full GRID control layer and without reliable central servers.

The interacting effects of problem difficulty, network size, and failure patterns on optimization performance and scaling behavior are still poorly understood in P2P global optimization. In this chapter we present empirical results comparing two P2P algorithms for real-valued search spaces in large-scale and unreliable networks: a distributed particle swarm optimization (PSO) algorithm [5] and a novel P2P branch-and-bound (B&B) algorithm based on interval arithmetic. Although our B&B algorithm is not a black-box heuristic, the PSO algorithm is competitive in certain cases, in particular, in larger networks. Some interesting, and perhaps counter-intuitive findings are presented as well: for example, failures in the network can in fact significantly *improve* the performance of P2P PSO under some conditions.

Related work. Related work can be classified as parallel optimization, P2P networking, and, very recently, the intersection of these two fields. We focus on this last category, mentioning that, for example, [34] is an excellent collection of parallelization techniques for various algorithms, and, for example, [32] is a useful reference for P2P computing in general.

In P2P heuristic optimization, proposed algorithms include [5,24,37]. They all build on gossip-based techniques [15,23] to spread and process information, as well as to implement algorithmic components such as selection and population size control. Our focus is somewhat different from [24,37] where it was assumed that population size is a fixed parameter that needs to be maintained in a distributed way. Instead, we assume that the network size is given, and should be exploited as fully as possible to optimize speed. In this context we are interested in understanding the effect of network size on performance, typical patterns of behavior, and related scaling issues.

In the case of B&B, we are not aware of any fully P2P implementations. The closest approach is [4], where some components, such as broadcasting the upper bound, are indeed fully distributed, however, some key centralized aspects of control remain, such as the branching step and the distribution of work. In unreliable environments we focus on, this poses a critical problem for robustness.

Contributions. Our contributions include (i) a completely decentralized self-organizing B&B algorithm, presented in Section 2.2, where no manager or master nodes are needed and (ii) a scalability analysis, presented in Section 2.3, with simulations of P2P networks of various sizes involving node churn, and with a comparison to a P2P PSO implementation, that is designed to operate under the same conditions.

2.2 The Algorithms

Our target networking environment consists of independent nodes that are connected via an error-free message passing service: each node can pass a message to any target node, provided the address of the target node is known. We assume that node failures are possible. Nodes can leave and new nodes can join the network at any time as well.

In the following we describe two algorithms that operate in such networking environments. Both assume that all nodes in the network run an identical algorithm; thus no special role exists such as a master or slave. This design choice increases both robustness to failure and scalability.

None of the algorithms contain special methods to deal with leaving or joining nodes. New nodes simply start participating after a default initialization procedure, and failing nodes are tolerated automatically via the inherent (or explicit) redundancy of the algorithm design, as we explain later.

The above self-organizing features are made possible via a randomized communication substrate both algorithms are based on. This is remarkable especially because the two algorithms are rather different, yet they are based on similar basic P2P algorithms and services. In this section we first briefly overview this communication substrate, and then we discuss the two algorithms. Only our novel B&B algorithm is discussed in full detail, as the other ideas are taken from previous publications.

2.2.1 Peer Sampling and its Applications

We assume that all nodes are able to send a message to a random node from the network at any time. This very simple communication primitive is called the *peer sampling service* that has a wide range of applications [17]. In this chapter we will use this service as an abstraction, without referring to its implementation; lightweight, robust, fully distributed implementations exist based on the analogy of *gossip* [17]. We note that one of the earliest approaches to use a similar service, called *newscast*, was the DREAM framework [3].

The algorithms below will rely on two particular applications of the peer sampling service. The first is gossip-based broadcasting, and the second is diffusion-inspired load balancing. In gossip-based broadcasting, nodes periodically communicate pieces of information they consider "interesting" to random other nodes. This way, information spreads exponentially fast. Several techniques exist to increase the efficiency and performance of the method [10].

In diffusion-based load balancing, nodes periodically test random other nodes to see whether those have more load or less load, and then perform a balancing step accordingly. This process models the diffusion of the load over a random network.

Although we do not discuss the implementation of these functions, it has to be noted that their communication cost is moderate: gossiping involves periodically sending small messages to random peers. The period of communication can be configured. In our case this period will be

in the order of a few function evaluations. For difficult realistic problems this results in almost negligible communication costs.

2.2.2 P2P PSO

Based on gossip-based broadcasting, a distributed implementation of a PSO algorithm was proposed [5]. Here we summarize the main ideas to make the report self-contained.

Particle swarm optimization (PSO) [21] is a nature-inspired method for finding global optima of functions of continuous variables. Search is performed iteratively updating a small number N (usually in the tens) of random "particles" (solutions). The attributes of a particle are its current position vector $\boldsymbol{x}_i$, the current speed vector $\boldsymbol{v}_i$, the optimum point $\boldsymbol{p}_i$ and its *fitness* value $f(\boldsymbol{p}_i)$. The optimum point of a particle is the "best" position vector the given particle has been at. The PSO also stores the global best position $\boldsymbol{g}$, which is the best solution found by any of the particles.

The iterative update step mention above is as follows, based on the current attributes of the particle and the global best solution:

$$\boldsymbol{v}_i = \boldsymbol{v}_i + c_1 * rand() * (\boldsymbol{p}_i - \boldsymbol{x}_i) + c_2 * rand() * (\boldsymbol{g} - \boldsymbol{x}_i) \quad (2.1)$$

$$\boldsymbol{x}_i = \boldsymbol{x}_i + \boldsymbol{v}_i \quad (2.2)$$

In these equations, $rand()$ is a random number from $[0, 1]$, while c_1 and c_2 are learning factors. A typical setting is $c_1 = c_2 = 2$. The speed is never increased over a maximum velocity $vmax_i$, specified by the user.

We have assumed so far that all particles are aware of the global best point $\boldsymbol{g}$. Effects of incomplete topologies, where particles are aware of only a limited number of neighboring other node's best solutions, have been studied for sociology-inspired small-world graphs [19] as well as other types of random graphs [22]. Such studies were motivated by the intuition that incomplete topologies may prevent the system from getting stuck in local optima, thereby improving solution quality. Dynamic topologies based on particle clustering have also been proposed [20] to induce a differentiated behavior among groups of particles having similar local best positions. Still, full information has generally been shown to outperform partial topologies [25], so instead of incomplete topology approaches, that would be natural in the P2P model, we aim at using gossip techniques to provide an approximation of the global best value for all particles.

To adapt the PSO algorithm for a P2P environment, at each node p, we maintain and execute a particle swarm of size k. Each node p maintains a *swarm optimum* $\boldsymbol{g}^p$, which is the best solution ever seen by the given node: either found by one of its own particles, or received from other nodes in a message. Clearly, different nodes could have different swarm optima. Each node executes the PSO algorithm on its own swarm.

So far, we have not defined how the nodes exchange information. We implement this function using an anti-entropy epidemic algorithm that works as follows: periodically, each node p initiates a communication with a random peer q, selected through the peer sampling service

described in previous sections. Node p sends the pair $\langle \boldsymbol{g}^p, f(\boldsymbol{g}^p) \rangle$ to q, i.e. its current swarm optimum and its function value. When q receives this message, it compares the swarm optimum of p with its own swarm optimum; if $f(\boldsymbol{g}^p) < f(\boldsymbol{g}^q)$, then q updates its swarm optimum with the received optimum ($\boldsymbol{g}^q = \boldsymbol{g}^p$); otherwise, it replies to p by sending $\langle \boldsymbol{g}^q, f(\boldsymbol{g}^q) \rangle$.

The rate r at which messages are sent by the anti-entropy algorithm is a parameter of the algorithm which represents a tradeoff between the communication overhead and the accuracy of the approximation of the overall global optimum at all nodes.

No special provisions are taken to deal with failures. If messages get lost, the spreading of information slows down but is not compromised. Nodes may be subject to churn as well without affecting the consistency of the overall computation, due to the robustness provided by the peer sampling service. Joining nodes initialize particles with a random position and velocity; as soon as they receive an epidemic message containing a swarm optimum and its evaluation, their own approximation of the global optimum is updated.

Here we will use a special case of this algorithm, where particles are mapped to nodes: one particle per node. The current best solution, a key guiding information in PSO, is spread using gossip-based broadcast. In a nutshell, this means we have a standard PSO algorithm where the number of particles equals the network size and where the neighborhood structure is a dynamically changing random network. For more details, the reader is kindly referred to [5].

2.2.3 P2P B&B

Various parallel implementations of the B&B paradigm are well-known [34]. Our approach is closest to the work presented in [8] where the bounding technique is based on interval-arithmetic [30]. The important differences stem from the fact that our approach is targeted at the P2P network model described above, and it is based on gossip instead of shared memory.

The basic idea is that, instead of storing it in shared memory, the lowest known upper bound of the global minimum is broadcast using gossip. In addition, the intervals to be processed are distributed over the network using gossip-based load balancing.

The algorithm that is run at all nodes is shown in Algorithm 1. Each node maintains a priority queue and a current best minimum value. The priority queue contains intervals ordered according to their lower bound, where the most promising interval has the lowest lower bound.

The lower bound for an interval is calculated using interval arithmetic, which guarantees that the calculated bound is indeed a lower bound. This way, in the lack of failures in the network, the algorithm is guaranteed to eventually find the global minimum. However, we continuously have a current best value as well, so the algorithm can be terminated at any time. Any function with a precise mathematical definition supports interval arithmetic, although in some cases at a relatively large cost. Detailed discussion of the details of interval arithmetic is out of the scope of this chapter, please refer to [30].

We start the algorithm by sending the search domain D with lower bound $b = \infty$ to a random node. In faulty environments we can send this initial interval to more than one node.

Algorithm 1 P2P B&B

```
1: loop                                                                  ▷ main loop
2:     I ← priorityQ.getFirst()          ▷ most promising interval; if queue empty, blocks
3:     (I_1, I_2) ← branch(I)                  ▷ cut the interval in two along longest side
4:     min_1 ← upperBound(I_1)          ▷ minimum of 8 random samples from interval
5:     min_2 ← upperBound(I_2)
6:     min ← min(min,min_1,min_2)                    ▷ current best value known locally
7:     b_1 ← lowerBound(I_1)          ▷ calculates bound using interval arithmetic
8:     b_2 ← lowerBound(I_2)
9:     priorityQ.add(I_1, b_1)                ▷ queue is ordered based on lower bound
10:    priorityQ.add(I_2, b_2)
11:    priorityQ.prune(min)          ▷ remove entries with a higher lower bound than min
12:    p ← getRandomPeer()                              ▷ calls the peer sampling service
13:    sendMin(p, min)                                  ▷ gossips current minimum
14:    if p has empty queue or local second best interval is better than p's best then
15:        sendInterval(p, priorityQ.removeSecond())    ▷ gossip-based load balancing step
16:    end if
17: end loop
18: procedure ONRECEIVEINTERVAL(I(⊆ D), b)
19:    priorityQ.add(I, b)          ▷ D ⊆ ℝ^d is the search space, b is lower bound of I
20: end procedure
21: procedure ONRECEIVEMIN(min_p)
22:    min ← min(min_p,min)
23: end procedure
```

Termination is not discussed here: since it is an any-time algorithm, as mentioned above, any suitable termination condition (time-based, quality-based, etc) is applicable, just like in the case of metaheuristics.

Note that there are countless points where the algorithm can be optimized and fine-tuned, such as the branching step, the upper bound approximation, the load balancing step, and so on. There are various techniques to optimize the bounding step as well, such as using derivatives, etc. We intentionally keep the most basic version, which in this form has very few parameters. One of them is the number of samples in the upper bound approximation that we fix at 8. The other is the number of nodes the initial problem is sent to at startup time. This will be 1 for networks without failures, and 10 for networks with churn (nodes joining and leaving).

2.3 Experimental Results

The algorithms described above were compared empirically using the P2P network simulator PeerSim [28].We first describe the experimental setup and methodology and subsequently we present and discuss results.

	Function $f(x)$	D	$f(x^*)$	K
Sphere2	$x_1^2 + x_2^2$	$[-5.12, 5.12]^2$	0	1
Sphere10	$\sum_{i=1}^{10} x_i^2$	$[-5.12, 5.12]^{10}$	0	1
Griewank10	$\sum_{i=1}^{10} \frac{x_i^2}{4000} - \prod_{i=1}^{10} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	$[-600, 600]^{10}$	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

Table 2.1: Test functions. D: search space; $f(x^*)$: global minimum value; K : number of local minima.

2.3.1 Experimental Setup

We selected well-known test functions as shown in Table 2.1. We included Sphere2 and Sphere10 as easy unimodal functions. 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. These two functions were designed to mislead local optimizers. Finally, Levy4 is not unlike Griewank10, but more asymmetric, and involves higher amplitude noise as well. Levy4 is in fact specifically designed to be difficult for interval arithmetic-based approaches.

We considered the following parameters, and examined their interconnection during the experiments:

- **network size** (N): the number of nodes in the network
- **running time** (t): the duration while the network is running. Note that it is is *not* the sum of the running time of the nodes. The unit of time is one function evaluation.
- **function evaluations** (E): the number of *overall* function evaluations performed in the network
- **quality** (ϵ): the difference of the fitness of the best solution found in the entire network and the optimal fitness

For example, if $t = 10$ and $N = 10$ then we know that $E = 100$ evaluations are performed.

Recall, that the simulated network consists of independent nodes that are connected only via an error-free message passing service. Messages are delayed by a uniform random delay drawn from $[0, t_{eval}/2]$ where t_{eval} is the time for one function evaluation. In fact, t_{eval} is considerable in realistic problems, so our model of message delay is rather pessimistic.

To simulate churn, in some of the experiments nodes are replaced with a certain probability (churn rate) with uninitialized nodes in any given fixed-length time. For simplicity we applied

one fixed churn rate: 1% of nodes are replaced during a time interval taken by 20 function evaluations. The actual wall-clock time of one function evaluation has a large effect on how realistic this setting is. In real P2P networks the observed churn rate is around 0.01% per second, corresponding to 2-3 hour uptimes on average [9]. In our setting we allow for 2000 function evaluations during average uptime, which maps to 5 seconds per function evaluation. If a function is faster to evaluate, our churn rate setting becomes more pessimistic.

To simplify discussion, we assume that the startup of the protocol is synchronous, that is, all nodes in the network are informed at a certain point in time that the optimization process should begin. The fine details of the startup process is out of the scope of this chapter, but even in the worst case, in the lack of synchrony and a priori knowledge at the nodes, gossip-based solutions can be applied that are orders of magnitude faster than the timescale of the optimization task.

2.3.2 Results and Discussion

Since neither of the algorithms is fine tuned, and since our focus is the exploitation of very large networks, here we are interested in understanding the overall scaling behavior of the algorithms. We focus on two key properties in this context: (i) scaling with the constraint of a fixed amount of available function evaluations, and (ii) with the constraint of having to reach a certain solution quality.

Our first set of experiments involves running the two algorithms with and without churn until 2^{20} function evaluations are consumed.[1] There are two questions one can ask: what is the solution quality that is reached, and what is the running time of the algorithms?

Solution quality is illustrated in Figure 2.1. The first clear effect is that for the larger networks, where network size is close to the available function evaluations, performance degrades quickly in all cases. This is not surprising, as in that case there are only very few evaluations available at all nodes, so search degrades to random search *if* the algorithm is greedy and wants to utilize all available resources as quickly as possible.

There is an interesting case though: Sphere2. It is not shown because it is so easy for both algorithms that it is solved to optimality by P2P B&B in all cases, and by PSO in almost all cases except for the largest networks. The interesting effect we can discover is that the B&B approach "refuses" to utilize the entire network, because it cannot generate enough promising intervals (pruning is "too" efficient) and therefore it can deliver optimal solutions irrespective of network size, but at the cost of longer running times (see also Figure 2.2, as explained later). Depending on the context, this effect can be very advantageous but harmful as well.

Another observation is that, while B&B is very efficient on the smaller networks, PSO consistently outperforms B&B on the large networks. Whereas B&B can never benefit from larger network sizes (since it only increases the chance of processing some intervals unnecessarily),

[1] We note here that for B&B, one cycle of Algorithm 1 was considered to take 20 evaluations, that is, in addition to the $2 \cdot 8 = 16$ normal evaluations, the interval-evaluation was considered to be equivalent to 4 evaluations (based on empirical tests on our test functions).

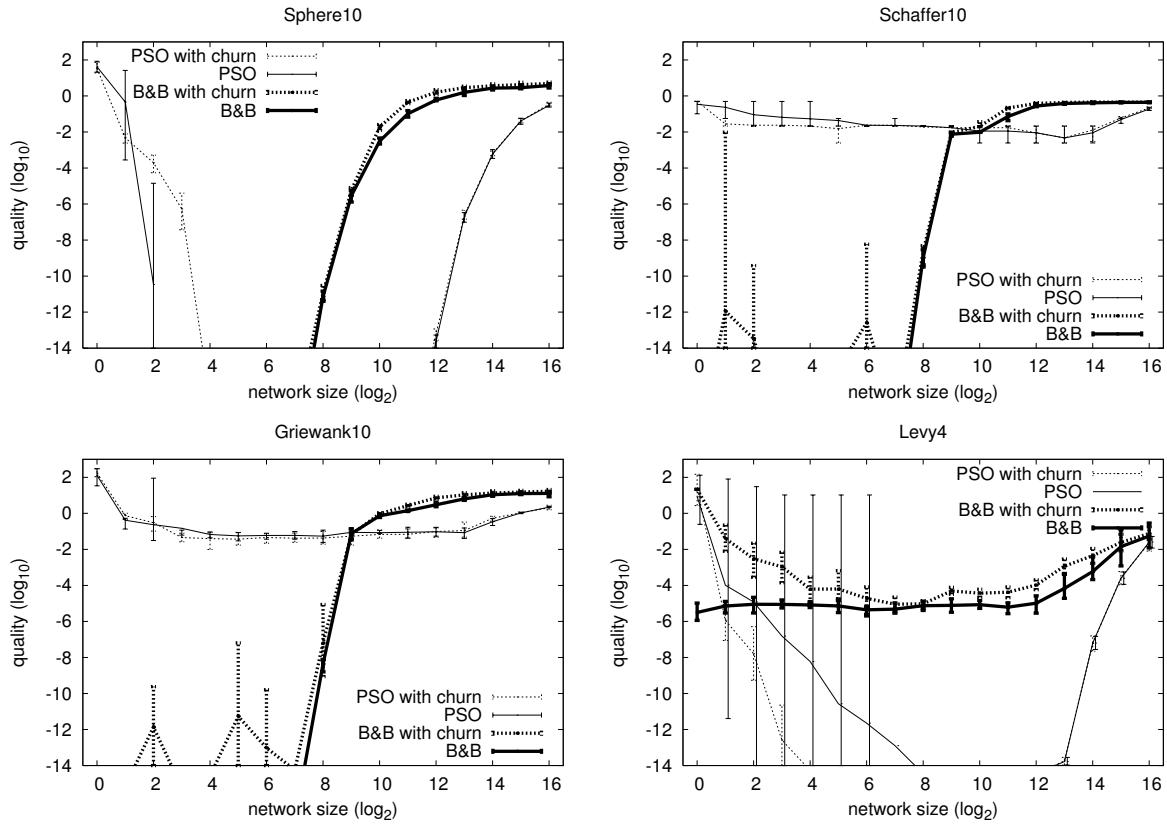


Figure 2.1: Solution quality as a function of network size and churn. Geometric mean is plotted (average digits of precision) with error bars indicating the 10% and 90% percentiles of 10 experiments (100 experiments for the more unstable Levy4). Note that lower values for quality are better (indicate more precise digits).

PSO has an optimal network size that represents enough possibility for exploration, but that also allocates enough function evaluations for each node to perform exploitation as well.

Function Levy4, that is hard for B&B, turns out to be easier for PSO, where PSO significantly outperforms B&B (note, that in the case of Griewank10 and Schaffer10, the situation is the opposite, and the sphere functions are easy for both algorithms). On Levy4, PSO does actually get stuck in bad local optima occasionally, but it can break out sufficiently often to provide a good average performance, whereas B&B gets bogged down not being able to do enough pruning due to the characteristics of the function.

A further support for this explanation is the curious effect of churn. On Levy4, churn increases the ability of PSO to break out of local optima via, in effect, restarting the nodes every now and then, while of course the global best solution never gets erased; it keeps circulating in the network via gossip. Indeed, in the experiments with churn, the performance of PSO is both better on average and more stable (has a lower variance). Again, just like larger networks, for B&B churn is always guaranteed to be harmful or neutral at best; Figure 2.1 also supports this

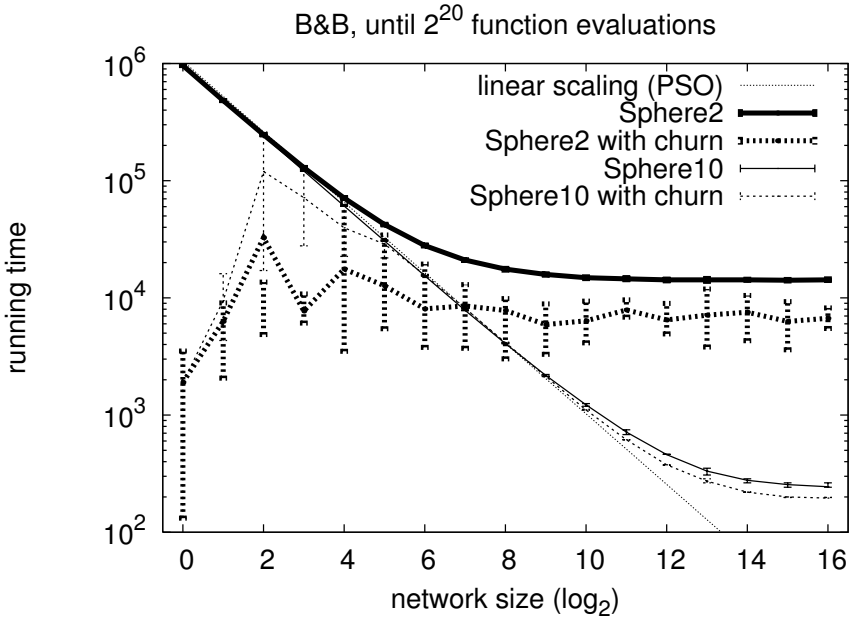


Figure 2.2: Running time of P2P B&B to reach 2^{20} evaluations. Average is shown with error bars indicating the 10% and 90% percentiles of 10 experiments

observation.

As of running time, P2P PSO always fully utilizes the network by definition, assuming a synchronous startup of the protocol, so its running time is $2^{20}/N$. This of course cannot be interpreted as linear scaling, since the actual quality of the output will be different in different network sizes.

In the case of P2P B&B, the situation is more complex, as illustrated by Figure 2.2. Only the sphere functions are shown: the other functions behave almost identically to Sphere10. For larger networks the curve leaves linear scaling since there startup effects start to become significant: B&B needs $O(\log N)$ cycles of its main loop (due to gossip-based load balancing) until sub-tasks reach all nodes. In addition, for problems that are especially easy, such as Sphere2, there are simply not enough sub-tasks to distribute because pruning is too efficient. This way, the algorithm never utilizes more than a given number of nodes, independently of how many are available. For more difficult problems, this effect kicks in at much larger network sizes.

In the case of churn, for small network sizes there is a significant probability that all nodes get replaced at the same time (that is, before old nodes could communicate with new ones). In such cases all sub-tasks get lost and the optimization process stops: hence the short running times, that do never reach 2^{20} evaluations.

Figure 2.3 shows results illustrating our second question on scaling: the time needed to reach a certain quality. The most remarkable fact that we observe is that on problems that are easy for B&B, it is extremely fast (and also extremely efficient, as we have seen) on smaller networks, but this effect does not scale up to larger networks. In fact, the additional nodes (as we increase

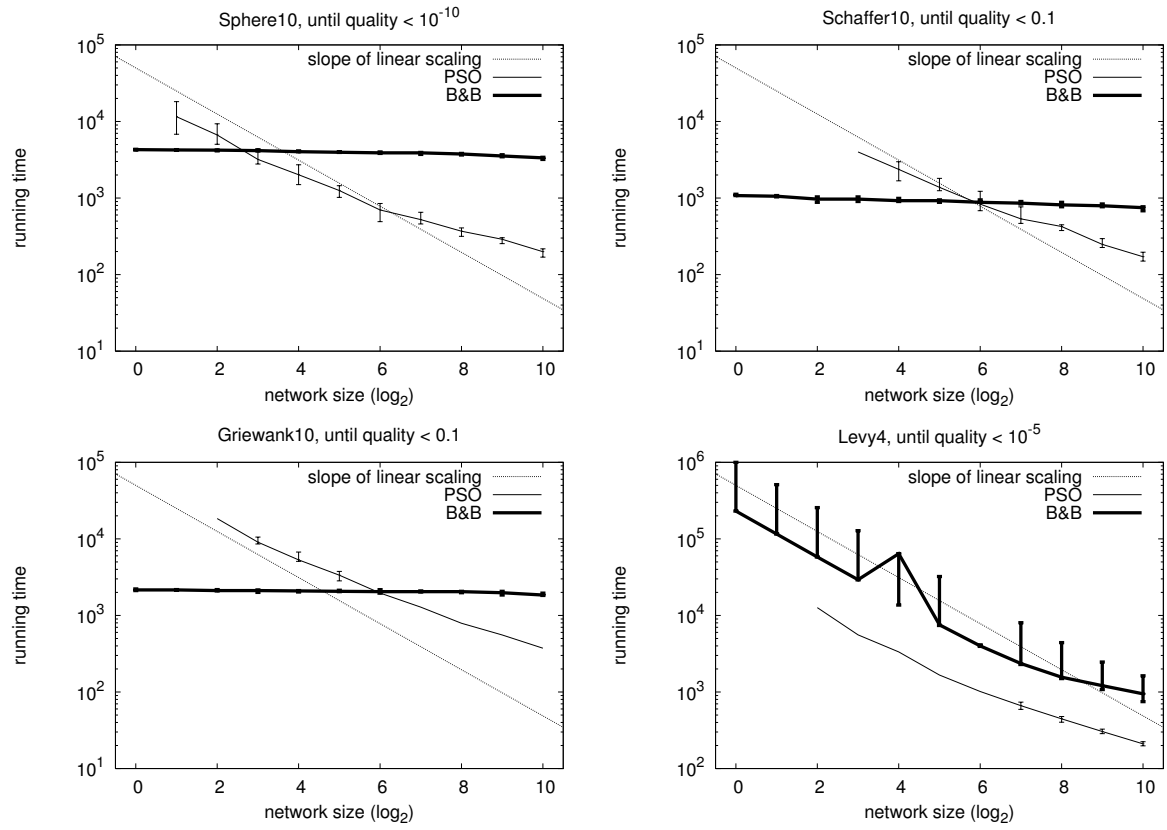


Figure 2.3: Running time to reach a certain quality (see plot titles). The median is plotted with error bars indicating the 10% and 90% percentiles of 10 experiments (100 experiments for the more unstable Levy4). Missing error bars or line points indicate that not enough runs reached the quality threshold within 2^{20} evaluations (when the runs were terminated).

network size) seem only to add unnecessary overhead. On Levy4, however, we observe scaling behavior similar to that of PSO.

2.4 Conclusions and Future Work

We have seen evidence that the set of difficult problems is different for the two algorithms tested, and overall they both show a rich set of behavioral patterns with respect to various aspects of scaling.

An interesting observation is that parameters of the environment, such as size and failure patterns should best be interpreted as (meta-)algorithm parameters controlling exploration and exploitation. In the case of B&B this meta-level operates on the intervals: if we increase network size, selection pressure decreases: more intervals get evaluated and "branched". However, since B&B is extremely conservative with the removal of intervals, decreasing selection pressure often results in increasing overhead, if the problem at hand is easy.

For P2P PSO, increasing the network size is equivalent to increasing the population size. Interestingly, a non-zero churn rate introduces a restarting operator for PSO, that can in fact increase performance on at least some types of problems.

Unlike in small-scale controlled environments, in P2P networks system parameters (like network size and churn rate) are non-trivial to observe and exploit. An exciting research direction is to monitor these parameters (as well as the performance of the algorithm) in a fully-distributed way, and to design distributed algorithms that can adapt automatically.

Chapter 3

Distributed Hyper-Heuristics

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 include 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 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 the scenarios and parameters.

3.1 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 [6, 11, 27].

HHs are similar to meta-heuristics; the difference 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. As opposed 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" [11] approaches while meta-heuristics can achieve better performance although 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 [31], 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 [26] and agent based approach is proposed that is nevertheless also conceptually centralized involving a single

HH agent. Finally, in [36] 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 [12], as well as the more established peer-to-peer [3] and Grid [18] 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 chapter 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 [10]. 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.

3.2 The Island Model

Our parallelization approach is based on a symmetric island model: we assume that we are given independent nodes, each of which running 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 all nodes can request a random node address from a local *peer sampling service* that returns a random sample from the entire network.

While this chapter does 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 can scale to millions of nodes [17]. From this point we simply assume that this service is accessible at all nodes. All the communication mechanisms we will define are based on gossip algorithms [10] 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 [14] to generate better neighborhood structures [7, 35]. For simplicity, in this chapter we opted for a random structure.

Independently of the algorithm run on the island, we always propagate the current best solution to all islands. This is also done through a gossip protocol: islands periodically send the best solution they know of to random other 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 [29].

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

3.3 Algorithms

3.3.1 The Basic Heuristics

Here we describe the set of algorithms $\mathcal{A}$ that our HHs will use to operate on. A typical HH takes low level operators often classified as simple hillclimbers and mutation operators [27]. 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 all HHs.

The set of algorithms is shown in Table 3.1. Heuristics A1-A6 are variants of differential evolution. We use standard notation as proposed in [33]. Due to the parallel setting, some explanations are in order. 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 interpreted to be local to the island. Heuristic A5 is like the "2" variants but using one random solution from the population and the global best; A6 is the isolated version of A5.

Algorithm A7 is described in [5] ans summarized in Section 2.2.2. It is a simple island-based PSO algorithm, assuming the best solution PSO relies on is the global best, propagated through 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 islands maintain a population of size 8. This makes it possible for a HH to change the algorithm while keeping the population.

3.3.2 Baseline HHs

We include in our pool two trivial HHs as a baseline. The first is called StatEq, a shorthand 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 all heuristics. 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 stick to it throughout the run (depending on network size, this introduces some variance).

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

3.3.3 Tabu

Our first non-trivial HH is adopted from related work [11]. Like the set of heuristics A1-A8, and all the rest of the HHs, this algorithm is run on all islands.

In the original sequential version, the basic idea is that Tabu maintains a tabu list of heuristics, and it also maintains a rank value for all heuristics, that can take an integer value from the interval $[0, |\mathcal{A}|]$. After running a heuristic, if it resulted in improving 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 [11]. Its maximal size is $|\mathcal{A}| - 1$, and operates in a first-in first-out fashion.

We parallelize this algorithm by running it on all islands, and 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).

3.3.4 SDigmo and DDigmo

A HH called Digmo was proposed in [36] designed for a local Grid environment within the European Space Agency. We adapted this method for our island model in two slightly different ways.

The basic idea of 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 algorithms, Digmo maintains a LIFO queue of size k, that contains the k last results of the algorithm (they propose $k = 5$). 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 to be proportional to $1/M_i$:

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

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

We adapt this algorithm to our island model through 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 LIFO 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 picking a heuristic at random using this distribution independently at all islands. We call this variant DDigmo.

In the static approach we still want the network to reflect the distribution P_i, however, we want to achieve this in a way that minimizes the number of islands that actually change their heuristic. 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 [16]. This protocol has 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. Its basic idea is simulating diffusion and thereby calculating averages, network size, and other statistics.

In SDigmo and DDigmo, based on extensive preliminary experiments, we fixed the parameters as $\alpha = 1$ and $k = 5$.

3.3.5 Pruner

The main motivation of applying HHs is arguably their ability to adaptively combine search diversification and intensification in order to produce good solutions. Nevertheless, 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 fits the problem at hand best, since meta-heuristics themselves could deal with balancing between exploration and exploitation to a certain degree, with different 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 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 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 through 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 shown in Algorithm 2. 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 not eligible any more, 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 happen, then nothing happens: the current algorithm is not changed.

It is important to note that—since all nodes manage their own eligible sets that can differ—Pruner can occasionally add a removed algorithm again if a result is received through gossip that ranks the given algorithm high enough.

3.3.6 Scanner

Apart from shrinking the eligible set in the same way as Pruner, the key idea of Scanner is giving a chance to all algorithms in order to get a more thorough picture of the performance of a given algorithm, and also to allow for possible synergic effects among the algorithms.

To achieve this, we implement two ideas. 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).

The Scanner HH is shown in Algorithm 3. 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

Algorithm 2 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
```

non-improving iterations for the current algorithm; and $phase$, a state variable that stores the current phase of the algorithm: SCAN or NORMAL. Function $\mathrm{MaxNonImproving}(phase)$ takes the phase as input and returns the maximum number of consecutive non-improving iterations any algorithm is allowed to take.

This hyper-heuristic 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 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, with the goal of verifying whether the new solution just

Algorithm 3 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
```

name	short description
StatEq	equal share for heuristics in space
DynEq	equal share for heuristics in time
Tabu	an island based version of [11]
SDigmo	static variant of the HH inspired by [36]
DDigmo	dynamic variant of the HH inspired by [36]
Pruner	focusing search on best heuristics
Scanner	attempting to give a chance to all heuristics

Table 3.2: Summary of our pool of HHs

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 $\mathrm{MaxNonImproving}(\mathrm{NORMAL}) = \lceil I/(c \cdot n_e) \rceil$, and $\mathrm{MaxNonImproving}(\mathrm{SCAN}) = \min(15, \mathrm{MaxNonImproving}(\mathrm{NORMAL})/2)$, where n_e is the size of the eligible set and c is the number of iterations since 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.

3.4 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 [28]. In the following we discuss the experimental setup and discuss the results.

3.4.1 Test Functions

We selected well-known test functions as shown in Table 3.3. We included Sphere10 as an easy unimodal function. Rosenbrock10 and Zakharov10 are included as non-trivial unimodal 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 more asymmetric, and involves higher amplitude noise as well.

Cassini1 and Cassini2 are realistic applications related to the Cassini spacecraft trajectory de-

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

sign 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 studied extensively and are known to contain an enormous number of local optima and to be strongly deceptive for local optimizers [1].

3.4.2 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, all islands are assigned an equal number of function evaluations: E/N.

We ran 10 independent experiments for all parameter combinations using

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

combined with all possible algorithms in Table 3.2 *and* the standalone versions of the algorithms in Table 3.1, on all test functions.

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

	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 3.4: Dominance matrix based on mean best fitness.

3.4.3 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 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 very realistic or interesting. 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.

From the remaining dataset, we filtered out those network sizes that, for a very similar reason, also hinder meaningful comparison: too large sizes ($N \geq 2^{14}$) allow too few evaluations per island even for 2^{20}, the largest value of E we used. Besides, in small networks ($N \leq 2^2$) the behavior of the algorithms is rather different, 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, these differences distort our conclusions as well.

3.4.4 Dominance Analysis

In the remaining data, we were interested in characterizing dominant protocols, that perform well in all cases. To achieve this, we calculated the dominance matrix as shown in Table 3.4. 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

	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 3.5: Mean best fitness rank statistics.

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 show ranking information regarding mean best fitness in Table 3.5. 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, etc.

These two tables together present 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. In fact, the second best heuristic, SDigmo, is also dominated by StatEq by a substantial margin: 34 to 22.

As a general pattern, we can observe that approaches that tend to be static and do not change the heuristic on an island 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 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 3.5 however, we can observe that A1 has the largest number of wins among the possible parameter settings. There is a catch though: its ranking distribution is bimodal: it

	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 3.6: Minimal best fitness rank statistics.

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 illustrated even better by Table 3.6 which, instead of the mean best fitness, is calculated based on the *best* result of the 10 independent runs: we can see that A1 can be very good, but this performance is very unreliable. The corresponding dominance matrix is shown in Table 3.7, where the best HHs have the same order, but A1 jumps ahead in dominance.

Of course, 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 have less weight in the comparison. On the dataset without the easy functions, we see a slightly changed dominance matrix (Table 3.8). Algorithm A1 now jumps back: in fact, it turns out A1 excels on the easy functions primarily. However, the best three HHs are still the same as in Table 3.4, which gives further evidence that a good HH can in fact achieve a better performance than any of the basic algorithms it is based on, and this performance is rater stable as well.

3.4.5 Statistical Tests

Before turning to a more fine-grained presentation of the performance of the algorithms, we first discuss here is whether the algorithms that have a similar dominance pattern are in fact

	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 3.7: 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 3.8: Dominance matrix based on mean best fitness, excluding Sphere and Zakharov from the dataset.

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 the question 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 [13], 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. Similarly, the difference between DDigmo and DynEq is not significant either. This is consistent with the very similar rank of these pairs in Tables 3.4 and 3.8.

For the rest of the algorithm pairs we cannot find another clear case where the difference could be questioned in general.

3.4.6 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 3.1 presents mean best fitness as a function of network size for the non-trivial test functions.

We can observe that StatEq is very stable and tends to be at the lower bound of the other algorithms (or even better than all, see Cassini1) except for a few special cases where A4 and Scanner show a good performance in a small region of the parameters.

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 could produce a better peak performance.

3.5 Conclusions

In this chapter we have presented convincing evidence through an extensive experimental analysis that a conceptually very simple baseline method is a very 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 in general, 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 mostly use as basic heuristics due to their competitive performance and simple configuration, strongly depend on the population distribution.

Note that in the usual sequential setting, that is, improving only one population (or solution) iteratively, being conservative is very difficult at best. Consequently, our results offer a new insight that could be useful even for sequential algorithms.

We also feel that, for example, SDigmo is very promising, outperforming both Scanner and Pruner, but more research is needed on why it did not turn out to be significantly different from StatEq, and whether there are problems or parameter settings where SDigmo is actually superior to StatEq.

[1] Cassini1$(-789.7652528252638,\ 158.30958439573675,\ 449.38588149034445,\ 54.713196036801925,\ 1024.7266958960276, 4552.859162641155) = 4.930707804754513$

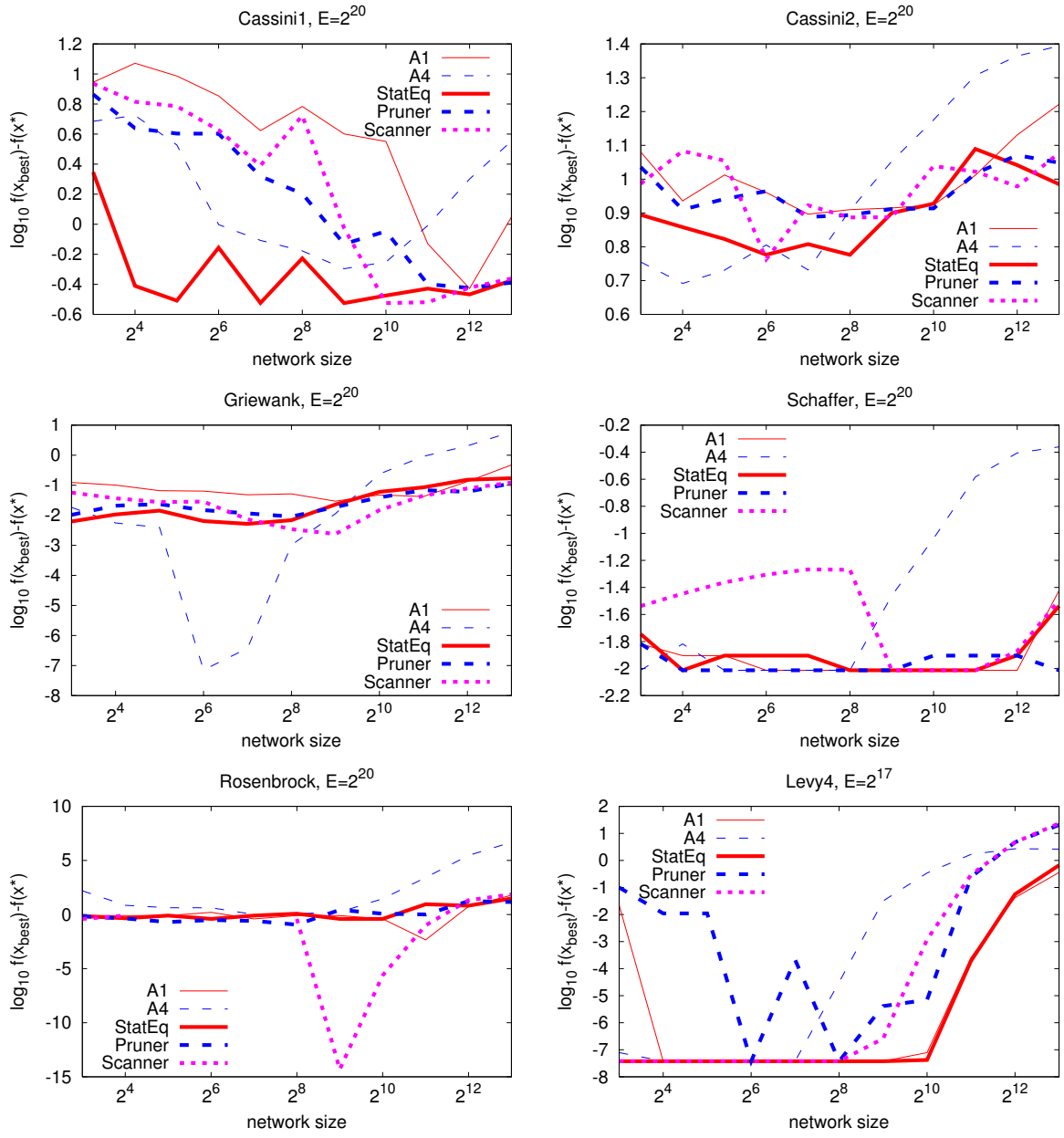


Figure 3.1: Mean best fitness (expressed as the difference from the optimal solution) on the non-trivial test functions as a function of network size.

Appendix A

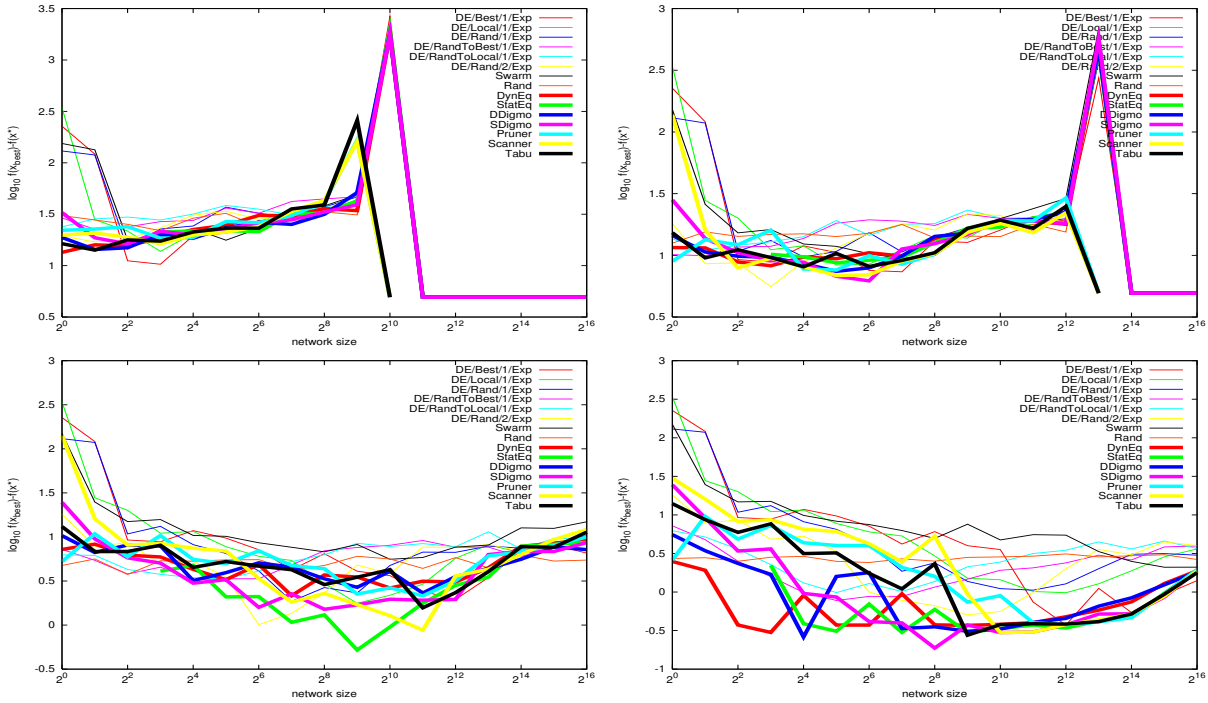
Complete Set of Results

In this appendix we present the complete set of results of the experiments presented in Chapter 3 in a raw form. As mentioned already, for all settings we ran 10 experiments. The first set of plots shows the mean best fitness, that is, the average of the outcome of these 10 experiments. The second set shows the minimum of these 10 experiments.

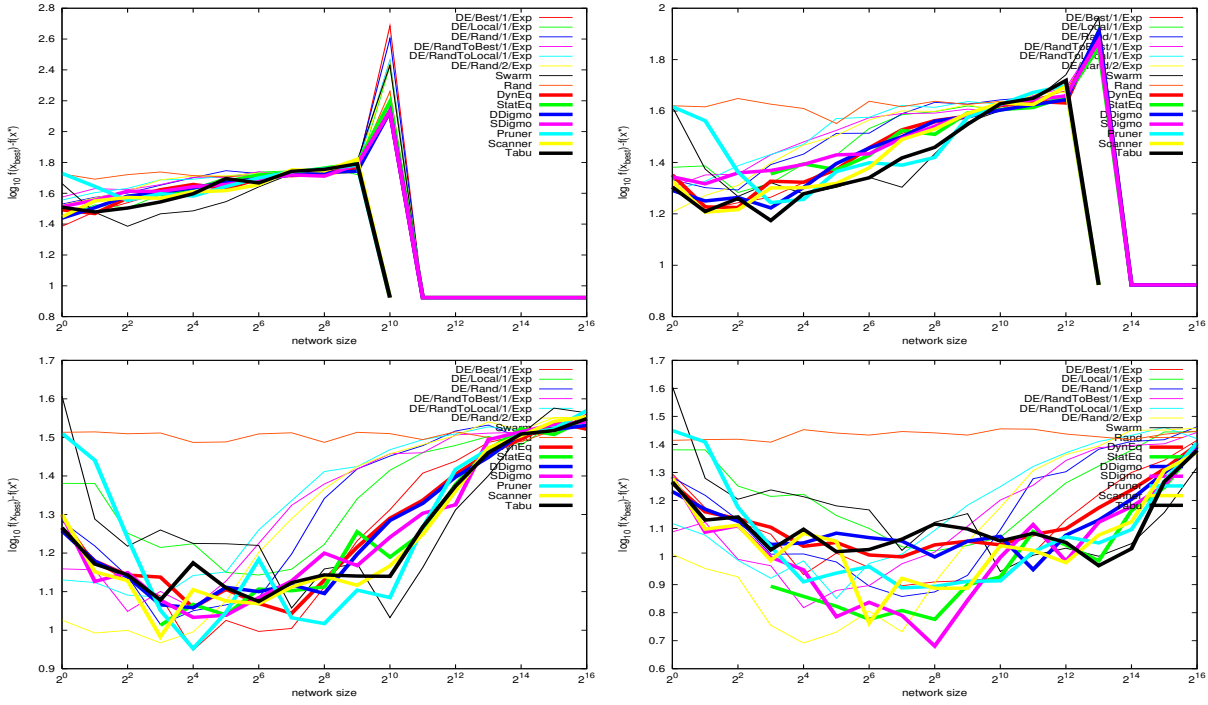
Under one heading the four plots correspond to $E = 2^{10}, 2^{13}, 2^{17}$ and 2^{20}, respectively.

Note that the plots are viewable only in high quality color print or on screen. A filtered and analised subset of these plots was presented in the previous chapter.

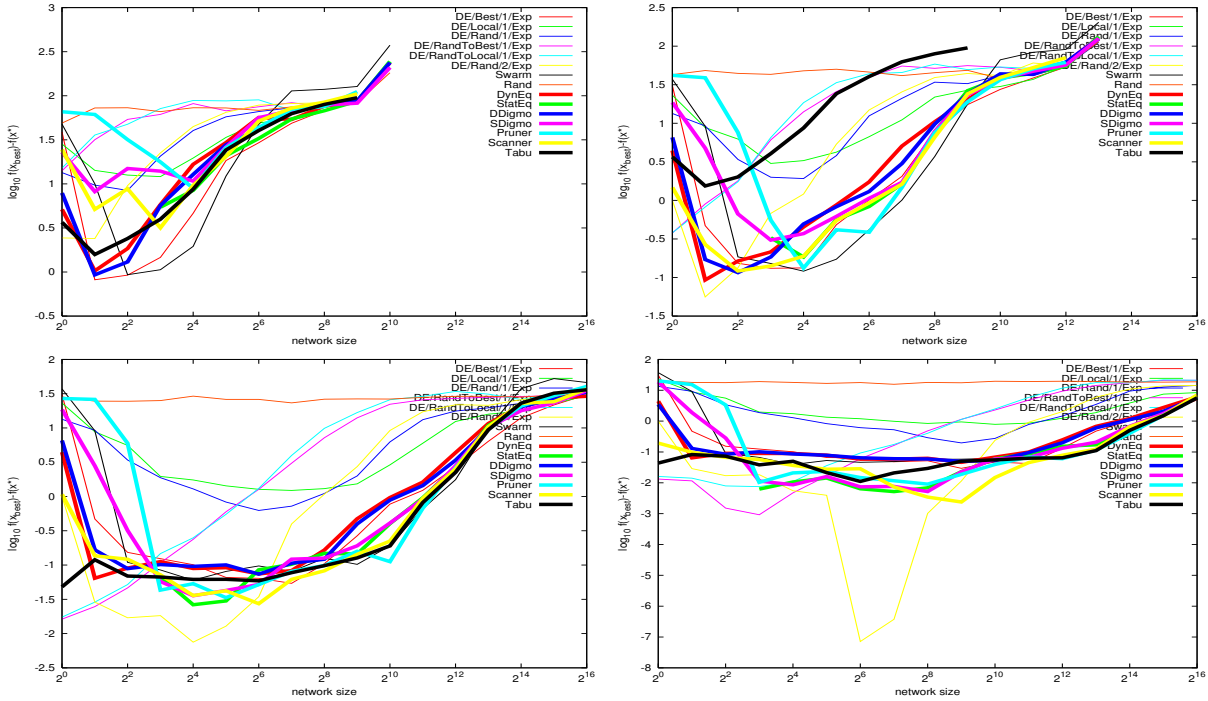
Cassini1



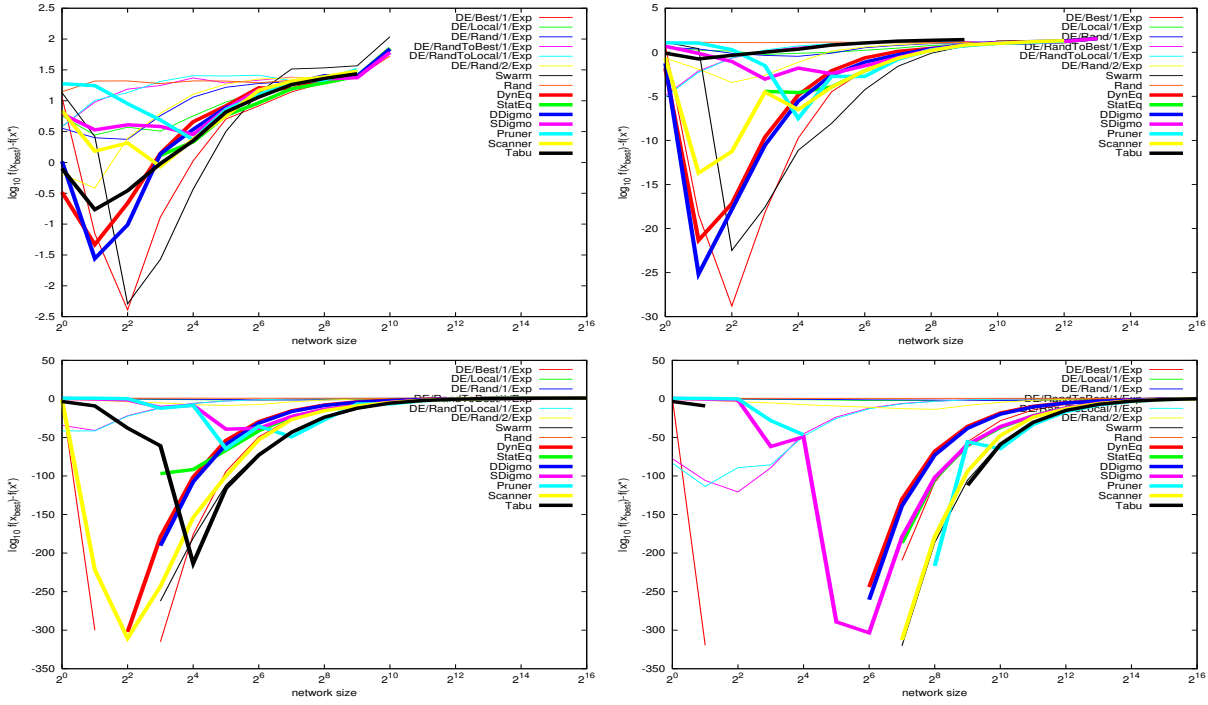
Cassini2



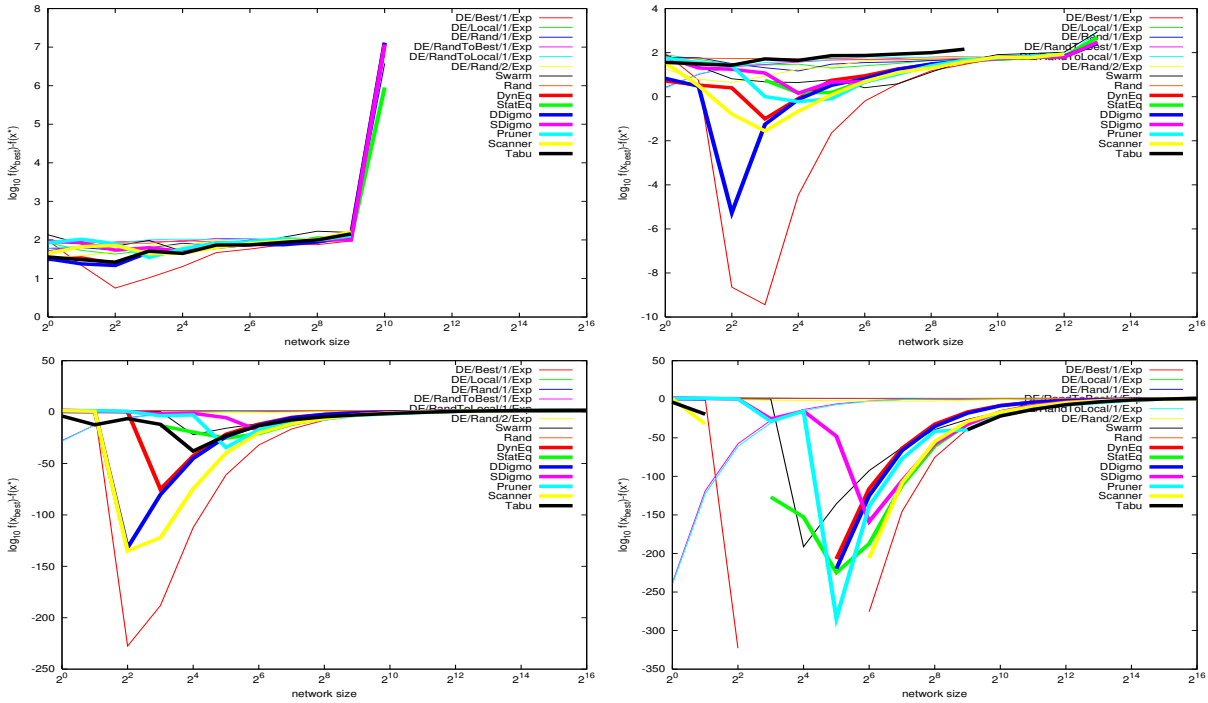
Griewank



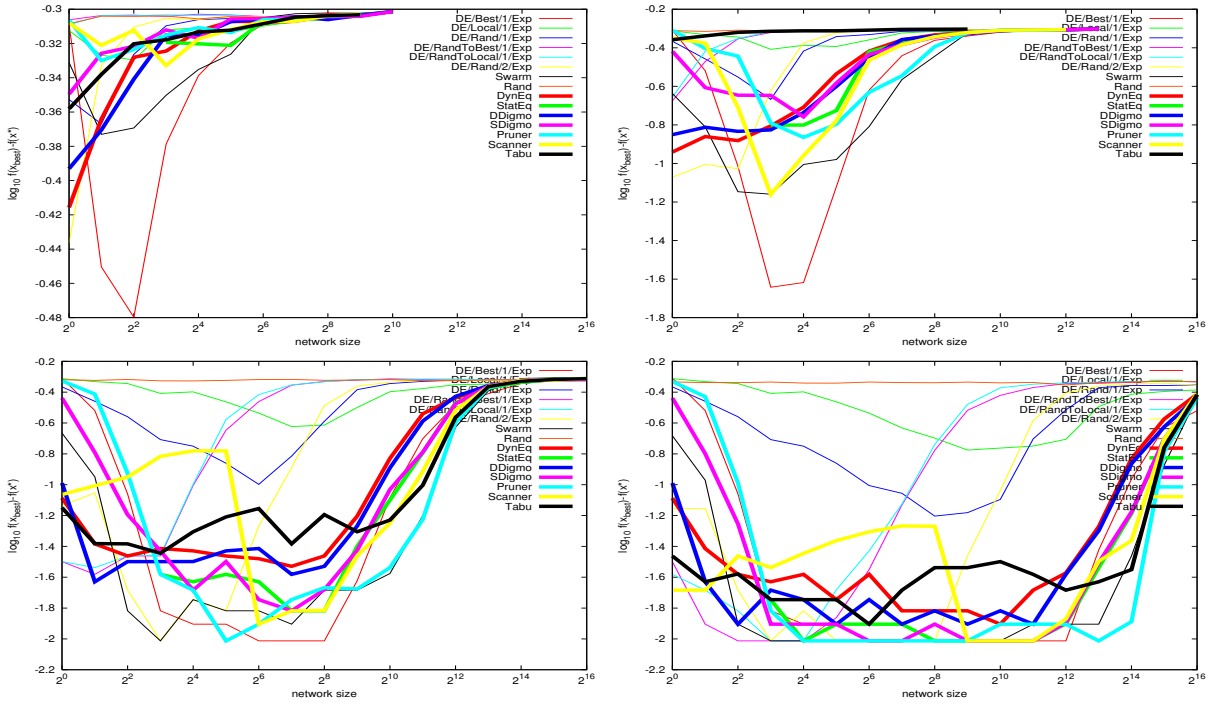
Sphere



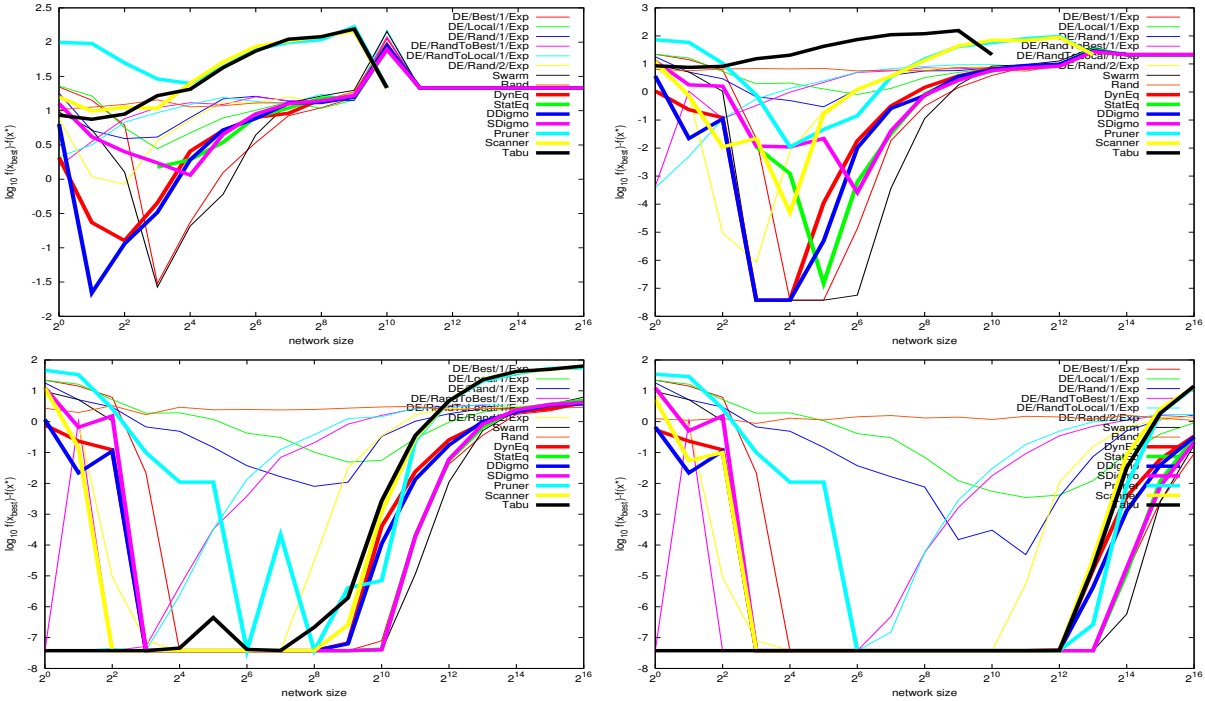
Zakharov



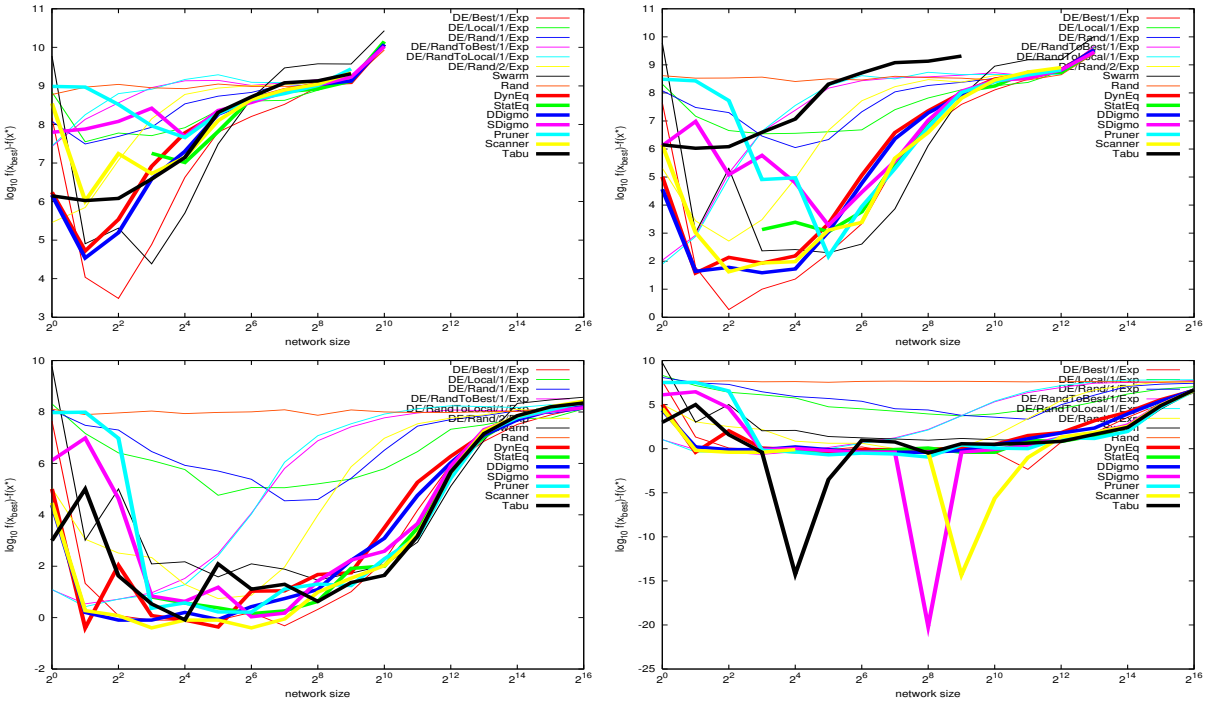
Schaffer



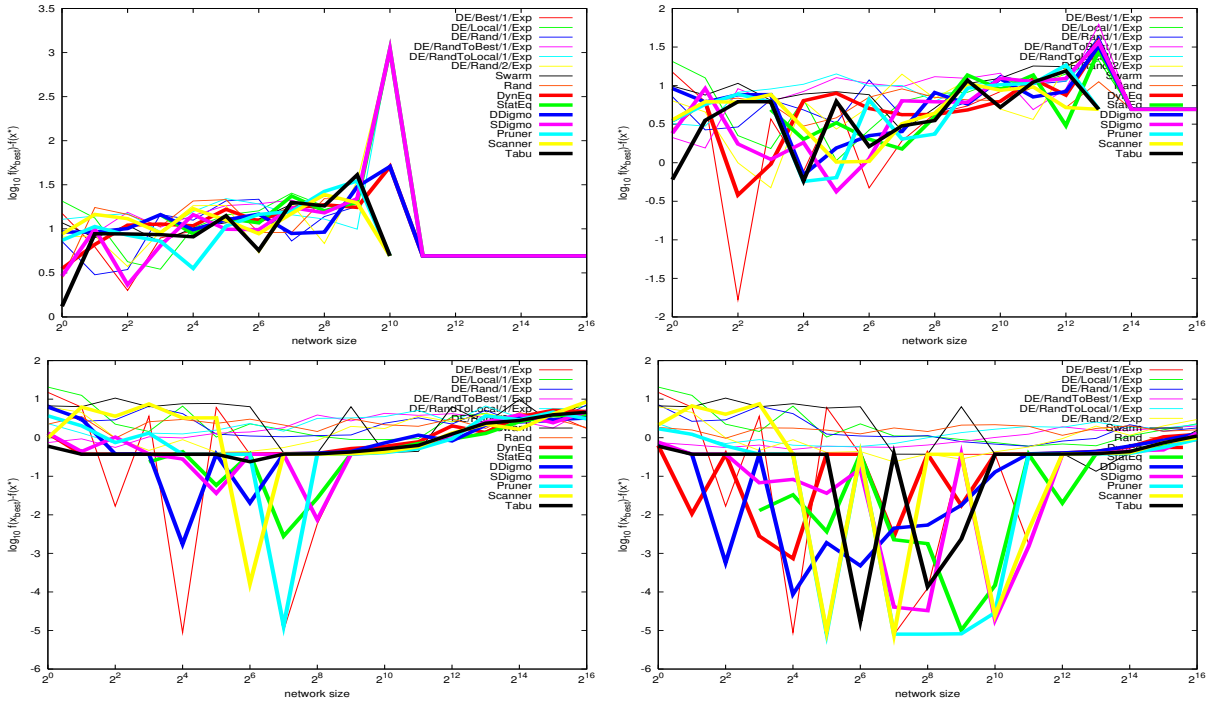
Levy4



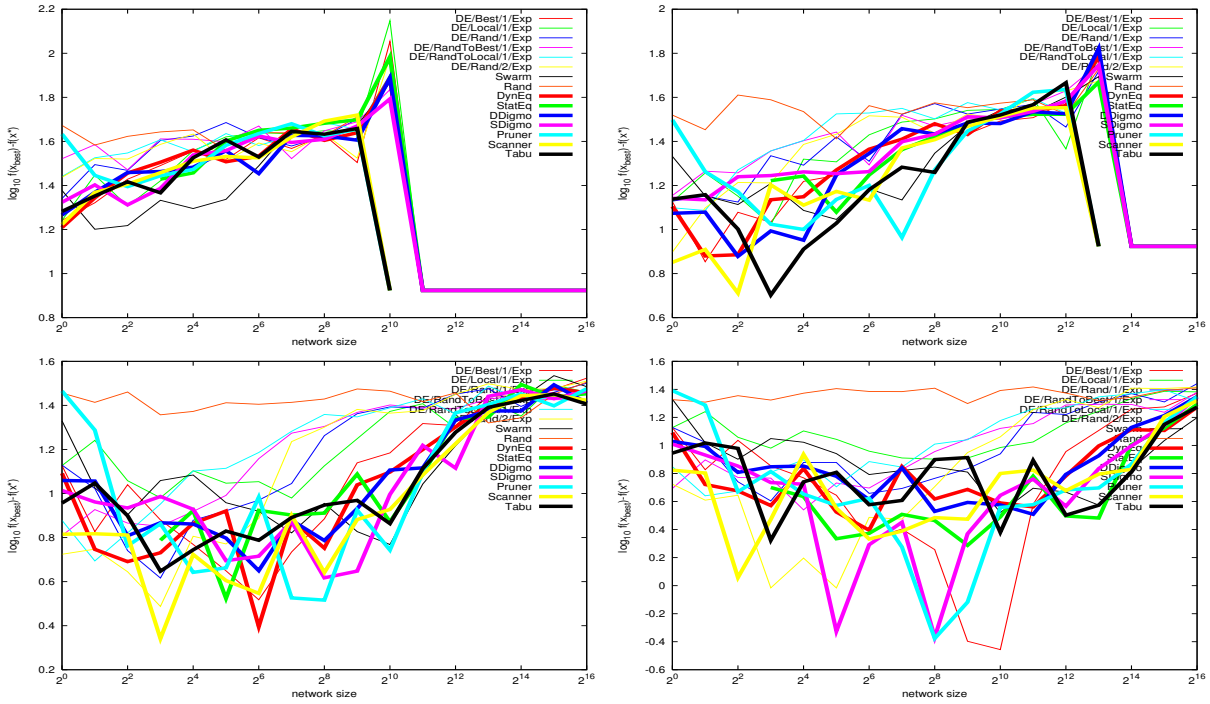
Rosenbrock



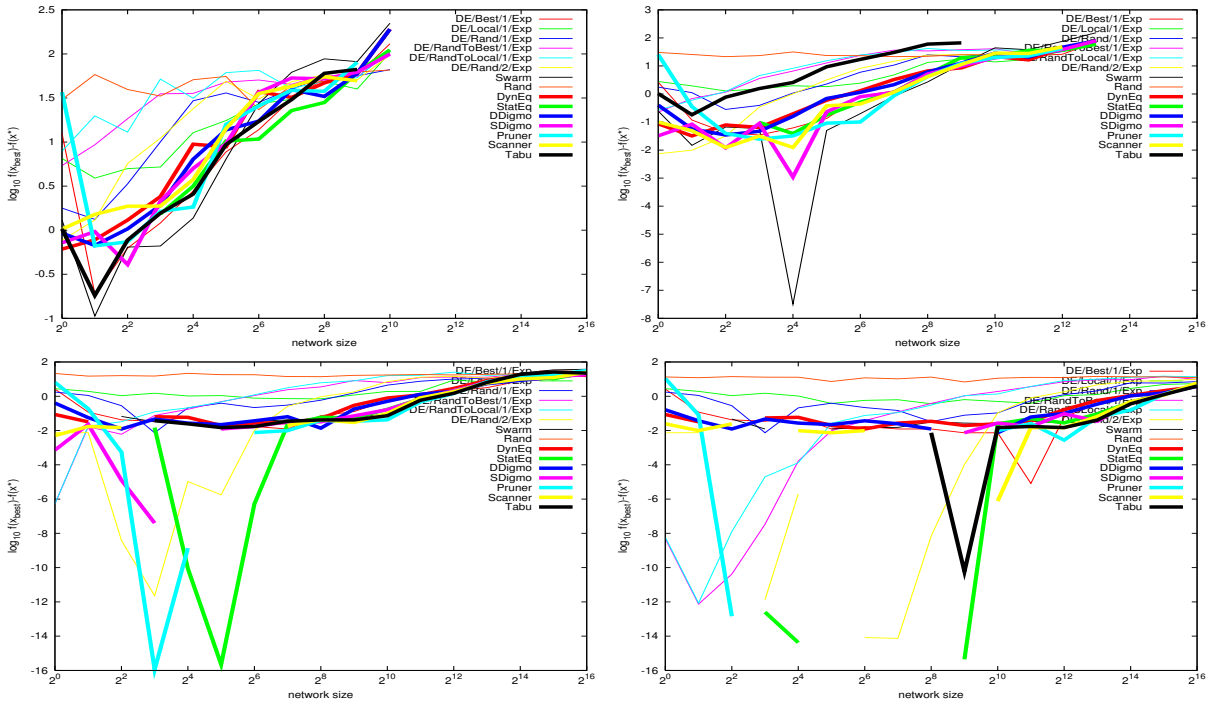
Cassini1 (min)



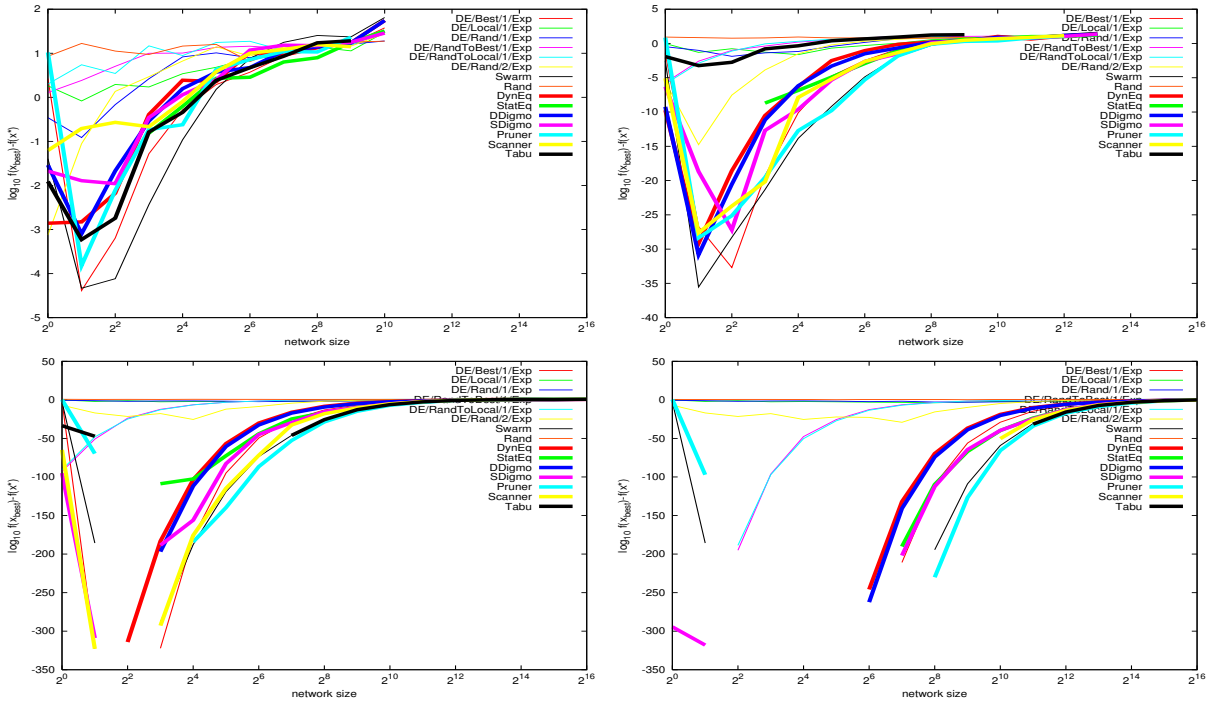
Cassini2 (min)



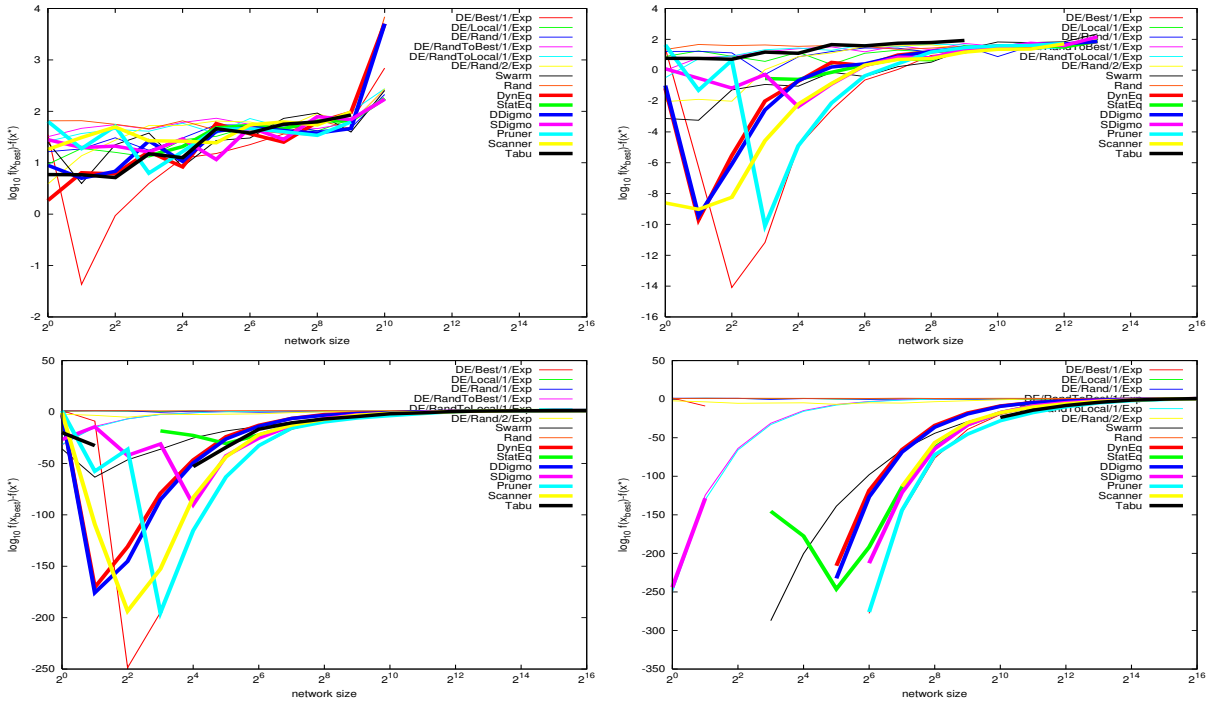
Griewank (min)



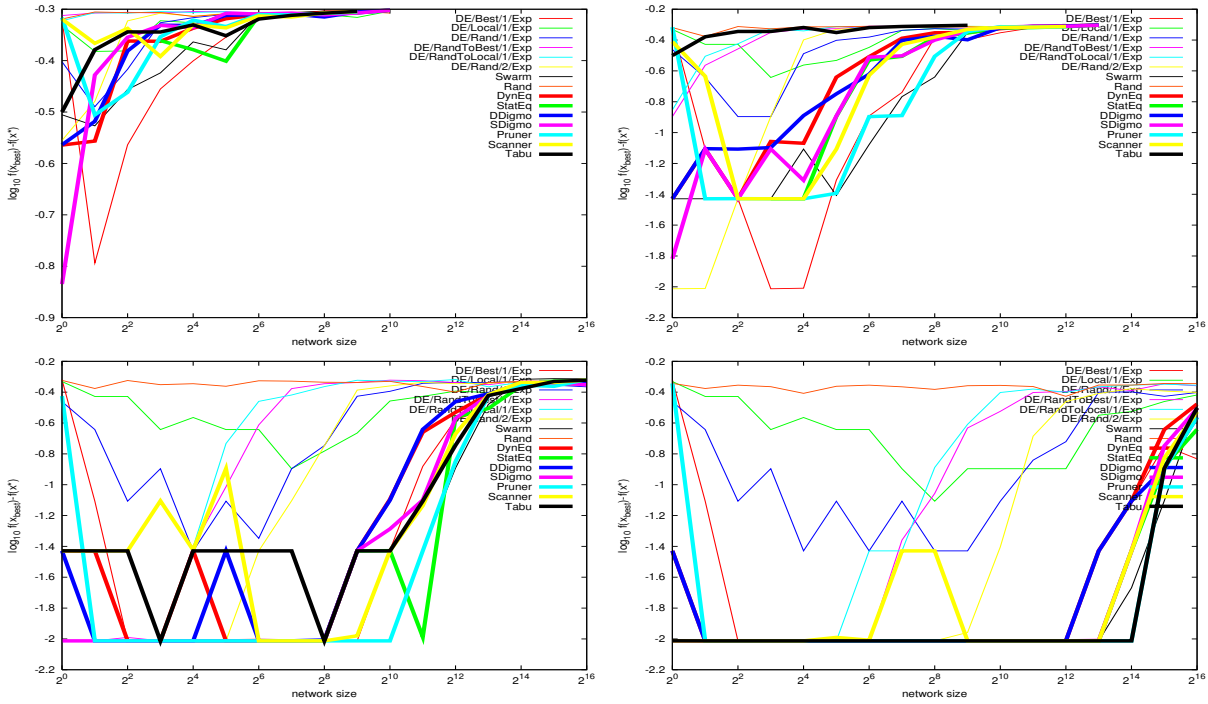
Sphere (min)



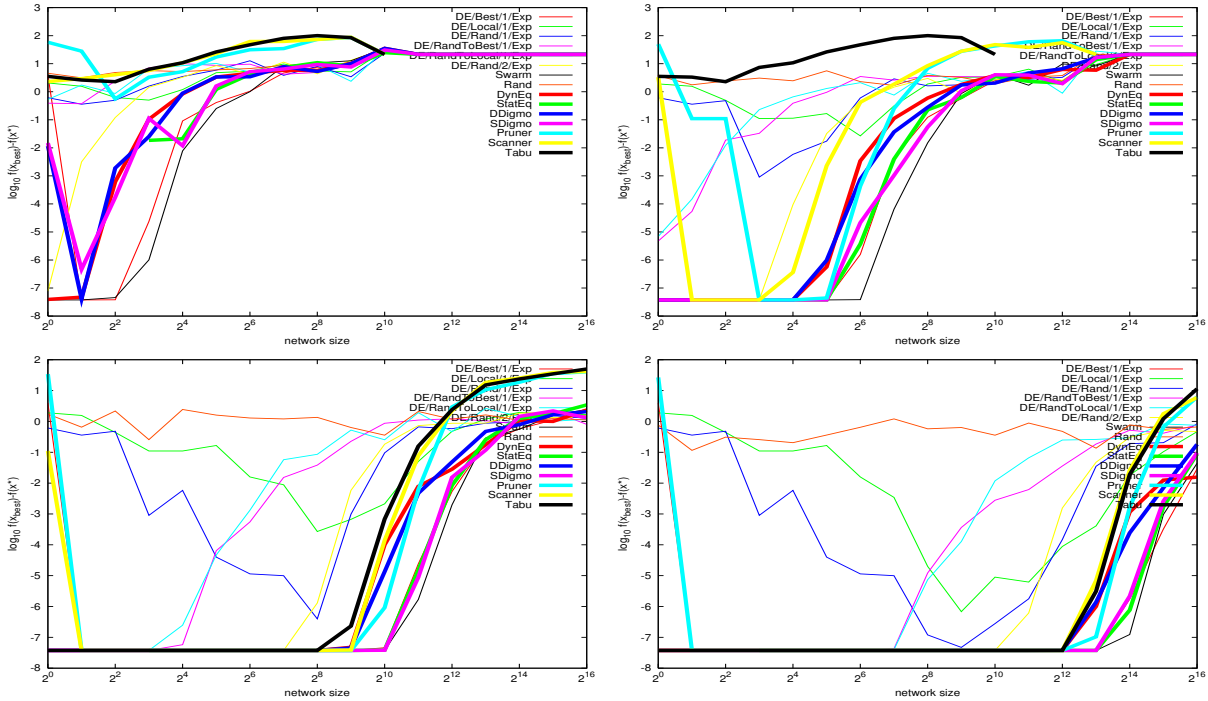
Zakharov (min)



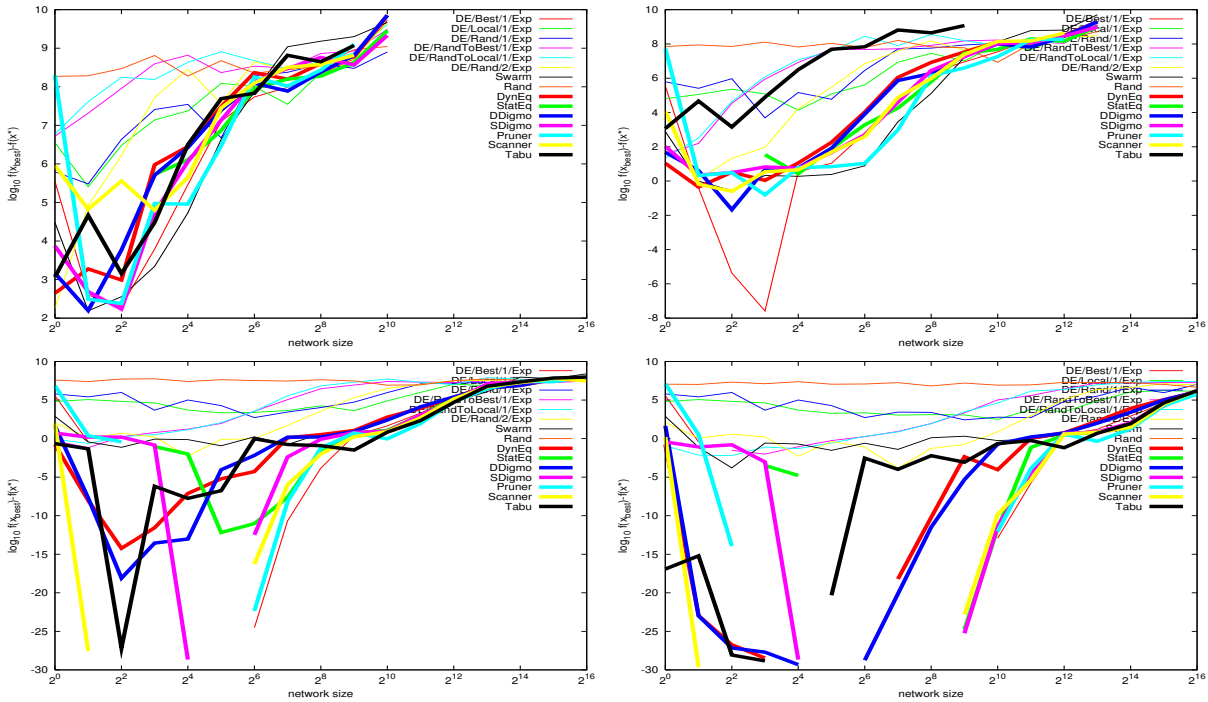
Schaffer (min)



Levy4 (min)



Rosenbrock (min)



Bibliography

[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-Interscience, 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 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 *Lecture Notes in Computer Science*, pages 665–675. Springer-Verlag, 2002.

[4] A. Bendjoudi, N. Melab, and E.-G. Talbi. A parallel P2P branch-and-bound algorithm for computational grids. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2007)*, pages 749–754, Rio de Janeiro, Brazil, 2007.

[5] 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, Apr. 2008.

[6] 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.

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

[8] L. G. Casado, J. A. Martinez, I. Garcia, and E. M. T. Hendrix. Branch-and-bound interval global optimization on shared memory multiprocessors. *Optimization Methods and Software*, 23(5):689–701, 2008.

[9] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*. IEEE Computer Society, 2004.

[10] 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, Vancouver, British Columbia, Canada, Aug. 1987. ACM Press.

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

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

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

[14] 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: Third International Workshop (ESOA 2005), Revised Selected Papers*, volume 3910 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2006.

[15] M. Jelasity, A. Montresor, and O. Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In G. Di Marzo Serugendo, A. Karageorgos, O. F. Rana, and F. Zambonelli, editors, *Engineering Self-Organising Systems*, volume 2977 of *Lecture Notes in Artificial Intelligence*, pages 265–282. Springer, 2004. invited paper.

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

[17] 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, Aug. 2007.

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

[19] J. Kennedy. Small worlds and mega-minds: Effects of neighborhood topology on particle swarm performance. In *Proc. 1^{st} Congress on Evolutionary Computation (CEC'99)*, pages 1931–1938, 1999.

[20] J. Kennedy. Stereotyping: Improving particle swarm performance with cluster analysis. In *Proc. 2^{nd} Congress on Evolutionary Computation (CEC'00)*, pages 1507–1512, 2000.

[21] J. Kennedy and R. C. Eberhart. Particle swarm optimization. *IEEE Int. Conf. Neural Networks*, pages 1942–1948, 1995.

[22] J. Kennedy and R. Mendes. Population structure and particle swarm performance. In *Proc. 4^{th} Congress on Evolutionary Computation (CEC'02)*, pages 1671–1676, May 2002.

[23] A.-M. Kermarrec and M. van Steen, editors. *ACM SIGOPS Operating Systems Review 41*. Oct. 2007. Special issue on Gossip-Based Networking.

[24] J. L. J. Laredo, E. A. Eiben, M. van Steen, P. A. Castillo, A. M. Mora, and J. J. Merelo. P2P evolutionary algorithms: A suitable approach for tackling large instances in hard optimization problems. In *Proceedings of Euro-Par*, volume 5168 of *Lecture Notes in Computer Science*, pages 622–631. Springer-Verlag, 2008.

[25] R. Mendes, J. Kennedy, and J. Neves. The fully informed particle swarm: Simpler, maybe better. *IEEE Transactions on Evolutionary Computation*, 8(3):204–210, June 2004.

[26] 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.

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

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

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

[30] H. Ratschek and J. Rokne. Interval methods. In R. Horst and P. M. Pardalos, editors, *Handbook of Global Optimization*. Kluwer, 1995.

[31] 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 Lecture Notes in Computer Science, pages 51–67. Springer, 2005.

[32] R. Steinmetz and K. Wehrle, editors. *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005.

[33] 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, Dec. 1997.

[34] E.-G. Talbi, editor. *Parallel Combinatorial Optimization*. Wiley, 2006.

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

[36] 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.

[37] W. R. M. U. K. Wickramasinghe, M. van Steen, and A. E. Eiben. Peer-to-peer evolutionary algorithms with adaptive autonomous selection. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1460–1467. ACM Press New York, NY, USA, 2007.