Evaluation of Seed Throwing Optimization Meta Heuristic in Terms of Performance and Parallelizability

Oliver Weede Stefan Zimmermann Björn Hein Heinz Wörn
Institute for Process Control and Robotics (IPR)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
e-mail: {oliver.weede, stefan.zimmermann2, bjoern.hein, woern}@kit.edu

***Abstract*— Seed Throwing Optimization is an easy to implement probabilistic metaheuristic for multimodal function optimization with roots in hill climbing and the evolutionary computation like technique Harmony Search. It is a randomized Gradient Ascent with multiple initial states and the possibility to limit exploration to only paths which have shown potential. In this paper, the speed of convergence of Seed Throwing Optimization is compared to Multi-Level Gradient Ascent, Harmony Search, Particle Swarm Optimization and Simulated Annealing. Improvements of the Seed Throwing Optimization are presented. Parallelizability of the mentioned metaheuristics is examined. Parts which are suitable for parallelization are extracted by identifying data and control flow dependencies. Two applications, port optimization in minimally invasive surgery and network parameter optimization for a distributed robotic system, are shown. The presented methaheuristics are tested in a benchmark. The highest convergence speed could be achieved with Harmony Search and Seed Throwing Optimization.**

Keywords-Multimodal function optimization; Parallel computing; Port optimization in minimally invasive surgery; Network optimization.

I. INTRODUCTION

Probabilistic metaheuristics are capable of solving very generalized classes of problems in many future computational applications. A metaheuristic is able to find a near optimum solution of a multi modal optimization problem in an efficient way. If a good solution of a multi modal function is found, it is not known if the solution is globally good. Thus the main problem that a metaheuristic has to deal with is to avoid premature convergence to a local optimum. If there is more than one good solution the issues arise of how to locate all (or some) of these solutions. There is a trade-off between *diversification* and *intensification*. Diversification refers to the exploration of the whole search space, intensification to the exploitation of the accumulated experience. The speed of convergence depends on the right balance of these forces [1].

Many approaches have been developed combining or enhancing existing methods, an overview and a taxonomy of global optimization methods can be found e.g. in Weise et al. [2], but there is no optimal solver for any optimization problem. Thus the best way is to use problem specific knowledge to choose an appropriate algorithm, convenient constraints to limit the search space or to modify strategies of an algorithm. Thus a further criterion of an optimization algorithm is its simplicity, which means that it is easy to understand and to implement.

A focus in this paper is Seed Throwing Optimization (STO), developed in 2009 [3]. It is a randomized Gradient Ascent with multi initial states and the possibility to explore only paths which have shown to be good. STO is a probabilistic metaheuristic based on the paradigm of an iterative local search. Thus, in each step a first solution is selected, a neighborhood of "similar" solutions is defined and the best solution of the neighborhood is determined. The idea of STO is to choose iteratively initial seeds over the domain of the objective function and to spread out more seeds in the neighborhood respecting the direction of the steepest ascent. Initial seeds are taken randomly or form a current best memory and are used for further exploration. The number of seeds "thrown out" from the initial point is controlled by the value of the function. Large values leads to extensive exploration. The neighborhood that is explored is like a gradient ray, but larger spreads have the form of a fan.

Parallelizability is another important criterion since clusters and multi-core processor architectures are commonly available and promise high acceleration. By splitting up algorithms appropriate high performance can be achieved. We compare parallelizability of several metaheuristics. In many real-life scenarios, objective functions are expensive to compute. In order to examine and classify STO in its potential parallelization degree, we have to identify parts of the algorithm that are suitable for parallelization. The pivot of successful parallelization is to avoid data and control dependencies between consecutive instructions that occur in the execution of the algorithm. For this, the parallelization of the algorithms will be discussed upon the pseudo code algorithm formulations introduced in this paper.

The objective function is treated as a black-box, no deeper insight of the characteristics is needed (direct optimization).

II. APPLICATIONS

We use STO in two applications that are shortly described in this section.

A. Port Optimization in Minimally Invasive Surgery

In minimally invasive surgery surgical instruments and an endoscopic camera are injected through so called trocars. The placement of the trocars is an important factor for the success of an intervention. The da Vinci® Surgical System

 ISBN: 978-1-61208-154-0

[4] is a telemanipulation system which is clinically used to perform interventions more ergonomic and precise. The manipulator is controlling the surgical instruments and the endoscope. Beside the trocar placement the selection of an appropriate initial pose of the manipulator is important. It contains the configuration of the stationary joints and the position and orientation of the manipulators base**.** Poor choices of the manipulators pose or the placement of the trocars can lead to collisions of the arms or unreachable target areas [12].

Finding optimal trocar positions and an optimal manipulator´s pose can be formulated as optimization problem consisting of active- and passive constraints and a goodness measure to be maximized. The passive constraint is corresponding to the trocar positions (ports), which remain stationary throughout the intervention. The pivot point of the surgical instruments has to be close to the chosen port positions. The active constraints refer to the trajectory of the end-effectors of the surgical instruments. Each target has to be reached. A configuration is rejected, if the passive constraints are not fulfilled (objective function is set to zero). If a target is unreachable or collisions occur the current configuration is rejected too. The objective function is a conjunction of several components. Separation of each arm to other arms, preoperatively segmented zones of risk and the bones are used as the main component of the objective function. Tilting of the table and ergonomic reasons like manipulation angle between the working instruments also influence this function. An evaluation of the objective function is very time consuming because minimal distances in a virtual scene containing over 200 000 triangles have to be computed for each point of the trajectory.

The result of the optimization is transferred to the operation theatre by using methods of augmented reality. The port positions are projected directly on the patient´s abdomen. The optimized pose of the manipulator is shown by a virtual scene containing a model of the manipulator and arrows to indicate the optimized pose.

B. Network parameter optimization for a distributed robotic system

State-of-the-art autonomous intuitive assistance and information systems for service robots compensate for the tremendous loss of information for remote users. Often augmented and/or virtual reality is used to provide immersion to the remote environment. A guaranteed minimum performance of the communication between the robotic system and the remote user is needed in order to still guarantee a suitable degree of immersion. Using non-dedicated standard means of communication, e.g. the Internet, makes constraints as audio and video synchronicity (<200ms) or latency for tactile tasks (<180ms) hard to fulfill [9]. A vast amount of multimedial and multimodal data has to be transmitted to provide a high degree of immersion. Thus, an optimization of the data flows in the robotic system as well as between the robotic system and the remote user is required to provide maximum immersion.

Based on the fact that there's no access to management functions of any transmitting node in the network for this scenario, two common approaches have been examined: (i) optimized individual data flows on the sending endpoints and (ii) traffic shapers on the sending endpoints.

In the first case, for every individual data stream measures as packet size and latency as well as global bandwidth and net load have to be optimized for a given network topology of a distributed robotic system and different kinds and severities of perturbations. In the second case, parameters for the traffic shapers like Hierarchical Token Bucket (priority, bandwidth, etc.) or Completely Fair Queuing have to be found. This is achieved by a near-reality test that runs the traffic of slightly modified original software components through the network simulator NS3 [11] that provides the internal network topology of the robotic system and the Internet as well as the perturbations.

The recorded sequence numbers and time stamps of the packages gained from the simulator run is then evaluated by an objective function script. The objective function is weighting the rate of arriving control commands (e.g. for robots) more than the rest of data streams (e.g. audio or video). The data stream throughput/quality and the inverse of the latency of packages also contributes to the objective function. If packages are lost the value of the objective function is decreased. These are the most important parameters which determine the configuration of the streams and/or the traffic shapers.

III. COMPARED METAHEURISTICS

In this section the five metaheuristics are shortly described.

C. Multi-level Gradient Ascent

Gradient Descent/Ascent figuratively speaking follows the graph from state to state, always locally increasing the value as much as possible by the updating rule

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\, \nabla \mathrm{f}(\boldsymbol{x}_k). \qquad (1)$$

The function could be treated as a black-box and finite differences could be used to approximate the steepest ascent. If the parameter $\boldsymbol{x}$ of the objective function has two dimensions, the function can locally be approximated by a plane. In this plane the steepest ascent can be computed.

Therefore, two points $\boldsymbol{b}$ and $\boldsymbol{c}$ in the local neighborhood of an initial point $\mathbf{a}=(a_1, a_2, a_3)^{\mathrm{T}}$ are chosen. Point $\mathbf{a}$ is the current best solution of the k-th iteration ($\mathbf{x}_k$ in eq. (1)). The three points determine the plane. With a parameter α in the real interval $[0, 2\pi]$ a circle on the plane is defined by the function

$$\mathbf{r}(\alpha) = (\mathbf{b} - \mathbf{a}) \sin \alpha + (\mathbf{c} - \mathbf{a}) \cos \alpha. \qquad (2)$$

To determine the steepest ascent, the first and the second deviation of the third component of $\boldsymbol{r}(\alpha)$ is regarded. By setting

$$\partial_\alpha\, r_3(\alpha) = 0 \text{ and } \partial^2_\alpha\, r_3(\alpha) < 0 \qquad (3)$$

 ISBN: 978-1-61208-154-0

```
(1)  Define f(x), x = (x1, x2, …, xN)^T
(2)  step width h_rought
(3)  infinitesimal step width h_inf
(4)  for all levels
(5)    h_rough = max( h_rough/2, h_inf)
(6)    select random solution x
(7)    p_start = GA(x, h_rough)
(8)    q = GA(p_start, h_inf)
(9)    remember q, if better
(10) end for
(11)
(12) function GA(starting point x_k, step width h)
(13) do
(14)   x_k+1 = x_k + h∇f(x_k)
(15)   evaluate function f(x_k+1)
(16) while (f(x_k+1) > f(x_k) or max. number of iterations reached)
(17) return x_k+1
```

Figure 1. Pseudo code of Multi-level Gradient Ascent (MLG)

the direction α of the steepest ascent in the plane can be computed. The solution for α is published in [3] (Eq. 4-7). With this method it is possible to approximate the gradient by only evaluating *one* further function value in each step. The points **b** and **c** out of step k can be used to define the plane for $\mathbf{x}_{k+1}$.

Gradient ascent only reaches an arbitrary optimum, not necessarily the global one. In *Multi-Level Gradient Ascent* several gradient ascents are performed with various initial states. Decimating the function in a way that only function values of a rough grid are chosen leads to a low pass filtering and many local optima are filtered out. In each level a gradient ascent is performed on the undersampled function and the result is used for a gradient ascent with a subtle grid. The mesh size of the rough grid is iteratively decreased. Fig. 1 shows the pseudo code of Multi-level Gradient Ascent.

D. Harmony Search

```
(1)  Define f(x), x = (x1, x2, …, xN)^T
(2)  accepting rate r_accept
(3)  pitch adjusting rate r_p
(4)  pitch variation range p_range
(5)  Generate best memory with B_size random solutions
(6)  while (maximum numbers of iterations not reached)
(7)    for all N components x_i
(8)      if (rand() < r_accept) choose value from best memory for x_i
(9)        if (rand() < r_p) x_i = x_i + r p_range with rand. val. r in [-1,1]
(10)     else
(11)       choose a random value for x_i
(12)     end if
(13)   end for
(14)   evaluate function f(x)
(15)   remember solution x in best memory, if better
(16) end while
```

Figure 2. Pseudo code of Harmony Search (HS)

Geem et al. [5] developed an optimization algorithm which is inspired by the improvisation process of a musician. Fig. 2 shows the pseudo code. rand() always denotes a function that returns equally distributed random numbers in the [0,1]-interval.

```
(1)  Define f(x), x = (x1, x2, …, xN)^T
(2)  depth of local search r_loc
(3)  maximum number of seeds thrown out n_MaxSeeds
(4)  Initialize best memory B with B_size random solutions
(5)  while (maximum numbers of iterations not reached)
(6)    if (rand() < 0.5) choose initial seed x_0 randomly
(7)    else choose seed x_0 from the best memory
(8)    endif
(9)    evaluate function f(x_0)
(10)   compute direction α of steepest ascent in x_0
(11)   compute number of seeds n_x (Eq. 5)
(12)   for each seed
(13)     compute position of seed x_i (Eq. 6)
(14)     evaluate function f(x_i)
(15)   end for
(16)   remember solution in best memory if better
(17) end while
```

Figure 3. Pseudo code of Seed Throwing Optimization (STO)

E. Seed Throwing Optimization

Seed Throwing Optimization (Fig. 3) has its roots in Gradient Ascent and Harmony Search.

For initializing the algorithm, the objective function f(**x**), $\mathbf{x} = (x_1, x_2, \ldots, x_N)^T$ and its domain is defined. The parameter B_{size} determines how many current best solutions are remembered in a (B_{size}, N)-matrix **B**. The size of the matrix should be chosen relatively small, thus a high intensification is ensured. The matrix is initialized with random values or, if known, with initial guesses for the solutions. The objective function is evaluated at these points. Associated to this matrix there is a B_{size}-vector containing the function values of the solutions. At every function evaluation during the runtime, the maximum and minimum values of the objective function are remembered within the variables g_{min} and g_{max}. The parameter r_{loc} controls the local search radius for initial seeds. We suggest taking the maximum interval of the domain divided by four. Choosing a big value leads to diversification, small values to intensification.

The main loop starts by selecting an initial seed $\mathbf{x}_0$. If a random number is below a threshold, an initial seed is chosen randomly; otherwise it is taken out of the matrix ***B*** which stores the best solutions. Like in Harmony Search this procedure could be done for each component of the solution to enable mutations of a known good solution. After evaluating f($\boldsymbol{x}_0$) an intensification factor n_x is computed. It determines how many seeds will be used to explore the neighborhood of $\boldsymbol{x}_0$. The higher the value f($\boldsymbol{x}_0$) the more seeds are used. Thus good solutions are explored more than valleys. We suggest to use

$$n_x = n_{MaxSeeds} \left\lfloor \frac{f(\mathbf{x}_0) - g_{Min}}{g_{Max} - g_{Min}} \right\rfloor . \qquad (5)$$

with the parameter $n_{MaxSeeds}$=5, which determines the maximal number of seeds "thrown out". The parameter n_x depends on the current maximum g_{max} and the current minimum g_{min} because the range of values of the objective function is not known. The parameter n_x is an intensification factor that controls the number of repetitions of the inner

loop (lines 12-15) of the algorithm. Two components of $\mathbf{x}_0$ are chosen randomly (e.g. x_1 and x_2). The direction of the steepest ascent for these chosen dimensions is computed (Eq. 4-7 in [3]). Let it be α_0, then $\mathbf{r}(\alpha_0)$ (Eq. 2) is the direction of the steepest ascent. Let $\mathbf{d}_\alpha$ be the normalized vector in this direction, and let $\mathbf{d}_\varphi$ be a normalized vector in a random direction φ. Than the position of seed i, which is "thrown out" from the initial seed, is determined by a convex combination of these vectors using two random numbers r_1 in [0.5,1] and r_2 in [0,1].

$$\mathbf{x}_i = \mathbf{x}_0 + r_2\, r_{loc}\, (\, r_1\, \mathbf{d}_\alpha + (1 - r_1)\, \mathbf{d}_\varphi\,). \qquad (6)$$

The function value of each seed $f(\boldsymbol{x}_i)$ is evaluated. If this solution is better than the worst solution in **B**, the new solution $\boldsymbol{x}_i$ replaces the old one and the new seed with its neighborhood has the potential to be further explored in a following iteration. If the maximum number of iterations is reached, the best solution of **B** is taken as the global best solution.

The performance can be slightly improved, if each time a new solution is entering the best memory a mutation of solutions is done. We suggest two ways of *mutation*:

Select the nearest solution $\mathbf{x}_b$ of the best memory to the new solution $\boldsymbol{x}_i$ in terms of Euclidean distance and evaluate the objective function at the convex combination of both solutions: Accept the resulting solution

$$\mathbf{x}_m = \frac{f(\mathbf{x}_b)\mathbf{x}_b + f(\mathbf{x}_i)\mathbf{x}_i}{f(\mathbf{x}_b) + f(\mathbf{x}_i)} \qquad (7)$$

if $f(\mathbf{x}_m)$ is better than the worst one in the best memory.

The other way of mutation is to evaluate the convex combination of all solutions of the best memory. Let $\mathbf{b}_1,\ldots,\mathbf{b}_{Bsize}$ be the solutions (rows) of matrix **B**. Then

$$\mathbf{x}_m = \frac{\sum_{i=1}^{B_{size}} f(\mathbf{x}_i)\mathbf{x}_i}{\sum_{i=1}^{B_{size}} f(\mathbf{x}_i)} \qquad (8)$$

replaces the worst solution, if its function value is better.

A possibility to further improve STO is to measure the similarity of a new solution to the most similar solution in the best memory in terms of Euclidean distance. If the distance is below a threshold, then the new solution should replace the most similar solution to ensure diversity of the best solutions.

In high-dimensional optimization problems the last dimensions which improved the global solution can be remembered and can then be preferred for further exploration, instead of randomly choosing two dimensions.

F. Particle Swarm Optimization

A description of Particle Swarm Optimization can be found in [6]. Fig. 4 shows the pseudo code.

```
(1)  Define f(x), x = (x1, x2, ..., xN)^T
(2)  Generate particles p_i with random solutions and velocity v_i=0
(3)  Initialize best solution for each particle p_best,i := p_i
(4)  Initialize best solution of all particles g_best
(5)  while (maximum numbers of iterations not reached)
(6)    for all particles i
(7)      compute vector to best solution of particle p_dir,i = p_best,i - p_i
(8)      compute vector to global best solution g_dir,i = g_best - p_i
(9)      determine random values r1, r2 in [0,1]
(10)     update velocity v_i = v_i + c1 r1 p_dir,i + c2 r2 g_dir,i
(11)     update solution p_i = p_i + v_i
(12)     evaluate function f(p_i)
(13)     remember best solution of particle p_best,i
(14)     remember global best solution g_best
(15)   end for
(16) end while
```

Figure 4. Pseudo code of Particle Swarm Optimization (PSO)

G. Simulated Annealing

Simulated Annealing is introduced in [7]. The pseudo-code is shown in Fig. 5.

```
(1)  Define f(x), x = (x1, x2, ..., xN)^T.
(2)  initial temperature T0
(3)  choose initial solution p and evaluate f(p)
(4)  step width h for neighborhood
(5)  while (t < maximum numbers of iterations)
(6)    choose point q in neighborhood of p
(7)    evaluate function f(q)
(8)    calc. current temperature temp, e.g. T0 c^t, 0<c<1 or T0 log(t+2)
(9)    while (max. num. of iters not reached AND q not accepted)
(10)     if (exp( f(q) – f(p)) / temp <= rand() or f(p) < f(q))
(11)       p=q (accept new solution q)
(12)     end if
(13)   end while
(14)   remember solution q if better than current best solution
(15) end while
```

Figure 5. Pseudo code of Simulated Annealing (SA)

IV. PARALLEL COMPUTATION TECHNIQUES

Although hardware models can be used to a great benefit for the practical application of parallelization, for our purpose, discussing the parallelization on a level of pseudo code algorithm formulations is sufficient. This way, the pivotal problem of identifying parts of the algorithms which can be computed parallel has to be solved and this favors brevity a lot.

As we assume no means to estimate the time required for an evaluation of the objective function, it has to halt for any parameter in the range examined in the optimization. The computing time spent executing the code for the optimization algorithms themselves is negligible in comparison to the computing time spent for the evaluation of the objective function.

Some work has been done in this area mostly for the popular Simulated Annealing and Particle Swarm

Optimization (PSO) algorithms. A taxonomy of parallelization techniques and their usage for Simulated Annealing as well as the effect on convergence caused by asynchronously parallelizing Simulated Annealing are discussed and presented in [8]. A simple approach to profit from wide-spread low-cost computing clusters using a Message Passing Interface (MPI) is shown in [8].

As the objective function is much more expensive to compute than the optimization algorithms themselves, only the ability to parallelize the objective function invocations is examined. The main goal is to fill the task pool sufficiently such that (i) the maximum of data-dependent tasks is available at the time and (ii) the data-independent tasks end about the same time, the data-dependency ends.

There are synchronous and asynchronous techniques for parallelization. Synchronous parallelization maintains the order of execution of a serial algorithm. Thus, synchronization points are required to ensure that all needed data is available. So workers (process internal or external threads) might wait for other workers to finish. It maintains the convergence behavior of the non-parallelized algorithms in terms of objective function calls. In contrast to synchronous parallelization, asynchronous parallelization does not force synchronization points that are blocking workers. It is favoring speed over using the most recent data from other workers and determinism to overcome Amdahls Law.

First, just the possibility for synchronous parallelization is described.

H. Synchronous Parallelization

Most time consuming parts of *Multi-level Gradient Ascent* are the function evaluations in the GD function (lines 12-17). There, two objective function evaluations (line 14 and 15) are needed in each iteration. These computations cannot be run in parallel since line 15 needs the result of line 14 and in the next iteration the last two solutions have to be known. Nevertheless, multiple levels running the GD function can be executed in parallel (line 4-10). The number of function evaluations in each level differs significantly, thus early finishing workers idle before completion of a level.

In every iteration of *Harmony Search*, a single new solution is generated (inner loop, lines 7-13) which is to be explored. Since this newly gained function value is compared to the values of the best memory each succeeding iteration depends upon the iteration before and therefore the iterations cannot be parallelized synchronously. Just the B_{size} solutions for initializing the best memory can be executed in parallel (line 5).

For *Simulated Annealing* just the first function evaluation (line 3) and the first evaluation of a new solution q (line 7) can be parallelized. Each further iteration (lines 5-13) depends on an initial state that is stored in the last iteration (line 11).

In *Particle Swarm Optimization* just the initialization of the particles (line 2) can be executed in parallel. If the number of particles is a multiple of the number of workers, the idle time of the workers is minimal. The particle loop (lines 6-15) cannot be executed in parallel by strict synchronous parallelization, because each particle can alter the current global best solution (line 12): to any of the lines considering the global maximum).

Seed Throwing Optimization has two major time-consuming code parts: (i) the ascent determining part in line 10 (computation of the steepest ascent) and (ii) the evaluation of the "thrown out" seeds before remembering best solutions and moving on to the next iteration (lines 12-15). In the ascent determining part three function evaluations can be run in parallel, and the function evaluations in the seed evaluation part can be run in parallel. It is ideal when the number of evaluated seeds is a multiple of the number of workers. Proceeding after each of these two parts requires all parallel function evaluations of the iteration to finish. Thus, calculating the function evaluation in parallel takes as long as the function evaluation with the longest computing time.

I. Asynchronous Parallelization

For increased speed-up, the parallel metaheuristics can be made asynchronous, by weakening the data dependencies by definition to avoid any synchronization delays. The workers then exchange needed data, which can be out-dated at the time they are used. The modified asynchronous algorithm could not only benefit from less synchronization delays but can even benefit from using its old data, as stated for Simulated Annealing in [7]. If this is the case, it indicates that there is no balance of intensification and diversification and the currently explored area is just a local maximum, not the global one. Nevertheless first experimental results have shown that using slightly obsolete data has little to no effect on the convergence of the algorithms.

In *Simulated Annealing* a point in the neighborhood is selected (line 6) and is then possibly accepted within a random walk (lines 9-13). One approach to parallelize the algorithm asynchronously is to run these steps in parallel by different workers. Since these steps are not deterministic new points are examined. In contrast, *Multi-Level Gradient Ascent* is deterministic and therefore does not examine new points. But Multi-Level Ascent can be parallelized asynchronously by starting work on a new level, each time a worker has finished a level.

PSO can easily be converted to an asynchronous version. Each time a particle has reached a new global optimum or its local optimum, the information is stored in shared memory. Thus, the particles can evaluate the function in parallel and information about the global maximum is, if so, just slightly obsolete. The same approach can be used for asynchronous *Harmony Search*. The best memory is stored in the shared memory, every time a better solution is found.

STO is very well suited for asynchronous parallelization due to the fact that potentially half the iterations use a random seeding point (line 6-8). These iterations are independent among themselves. The only dependency exists to the other half of the iterations using best memory-dependent seeding points. Thus, while providing an update attempt for the best memory for about every second iteration of the data-dependent case, the convergence of asynchronous STO is very much the same as serial STO. Each time a worker idles a random seeding point case can be performed.

 ISBN: 978-1-61208-154-0

V. EVALUATION

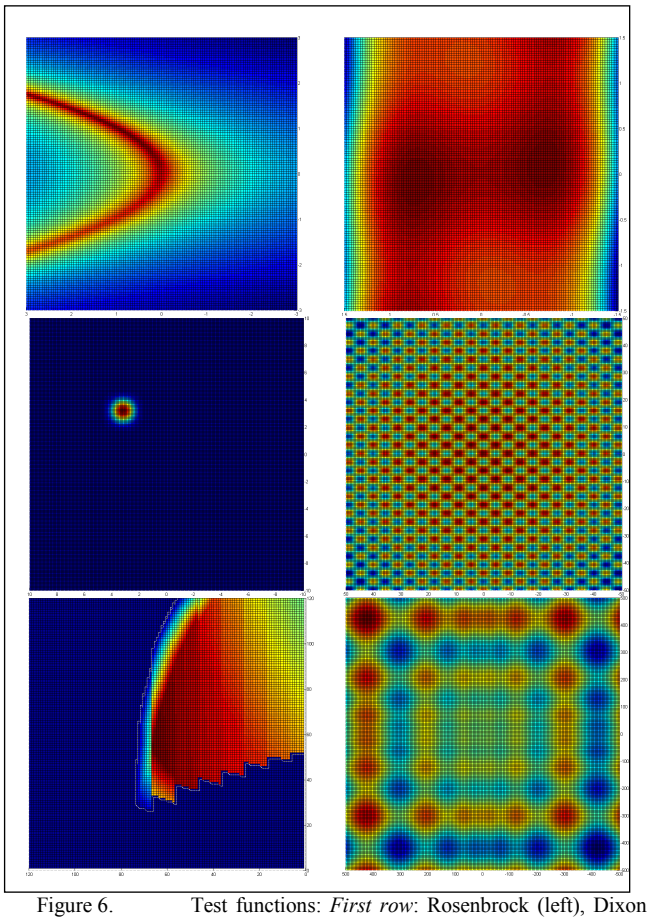

Figure 6. Test functions: *First row*: Rosenbrock (left), Dixon (right); *Second row*: Easom, Griewangk; *Third row:* Port, Schwefel. Maxima are colored in red.

An overview over the test functions, Rosenbrock´s Banana function, Dixon´s Camelback, Easom´s function, Griewangk´s and Schwefel´s function, can be found in [10]. We also tested the port optimization problem in minimally invasive surgery for the right-hand instrument. Fig. 6 shows the test functions and Table I the formulas and domains of the functions.

TABLE I. TEST FUNCTIONS, NAME, DOMAIN

$-\log\left(1+(1-x_1)^2+100\left(x_2-x_1^{\,2}\right)^2\right)+10$	Rosenbrock $[-3, 3]^2$
$-\left(4-2.1x_1^{\,2}+\frac{x_1^{\,4}}{3}\right)x_1^{\,2}-x_1x_2-4\left(x_2^{\,2}-1\right)x_2^{\,2}$	Dixon $[-1.5, 1.5]^2$
$\cos(x_1)\cos(x_2)\exp\left(-(x_1-\pi)^2+(x_2-\pi)^2\right)$	Easom $[-10, 10]^2$
$-\frac{x_1^{\,2}+x_2^{\,2}}{4000}+\cos(x_1)\cos\left(\frac{1}{\sqrt{2}}x_2\right)+3$	Griewangk $[-50, 50]^2$
$x_1\sin\left(\sqrt{\lvert x_1\rvert}\right)+x_2\sin\left(\sqrt{\lvert x_2\rvert}\right)$	Schwefel $[-500,500]^2$

Fig. 6 shows the test functions and Table I the formulas and domains of the functions. Each algorithm was run on each objective function 500 times. We are always looking for maxima of the objective functions. All tested objective functions have two dimensional solutions. Each algorithm is tested with several parameters.

As "bad anchor" random sampling (RND) of the functions while remembering the best function value was added as "optimization algorithm".

We tested the number of function evaluations to reach the 95% mark of the global maximum. In Fig. 7 the mean number of function invocations of all repetitions is shown.

Fig. 8 shows the best function value that is reached after 500 function evaluations. In this figure the median of the function values of all of experiments is shown as a percentage of the global maximum. The figure also shows the 0.975 and 0.025 quantiles. Thus, in 95% of all cases the function value is inside the shown interval. We choose the median and quantiles instead of mean and two standard deviations, because the data is not normally distributed.

In both figures (Fig. 7, 8) the results of the best parameters for each algorithm is shown.

In all experiments (except Random Sampling on Easom and Simulated Annealing on Schwefel) the 95% mark was reached (much) faster than 500 function evaluations. Thus, the results illustrated in Fig. 7 are important for applications in which a fast approximate solution is sufficient, and the results in Fig. 8 are more important if an accurate solution is needed.

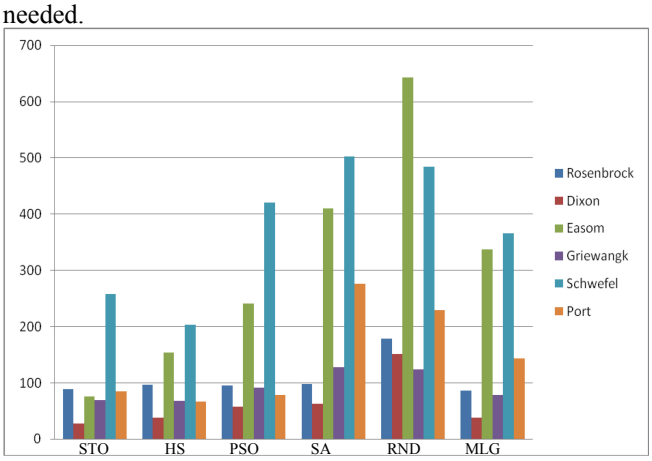

Figure 7. Mean convergence speed of Seed Throwing Optimization (STO), Harmony Search (HS), Particle Swarm Optimization (PSO) . Simulated Annealing (SA), Random Sampling (RND) and Multi-Level Gradient Ascent (MLG). Ordinate: Number of function evaluations to reach 95% mark of global maximum. Benchmark repeated 500 times.

STO was tested with best memory size three and five. There was no significant difference. In all test functions except for Rosenbrock´s function the results were better with $n_{MaxSeeds}$ = 5 than with $n_{MaxSeeds}$ = 10 or $n_{MaxSeeds}$ = 15. In the case of Rosenbrock´s function the results were achieved by choosing $n_{MaxSeeds}$ = 15.

We tested STO without gradient information and were only choosing random directions for "thrown out" seeds. In Rosenbrock´s and in Griewangk´s function the results were better. The "sharp edges" in these functions make it hard to

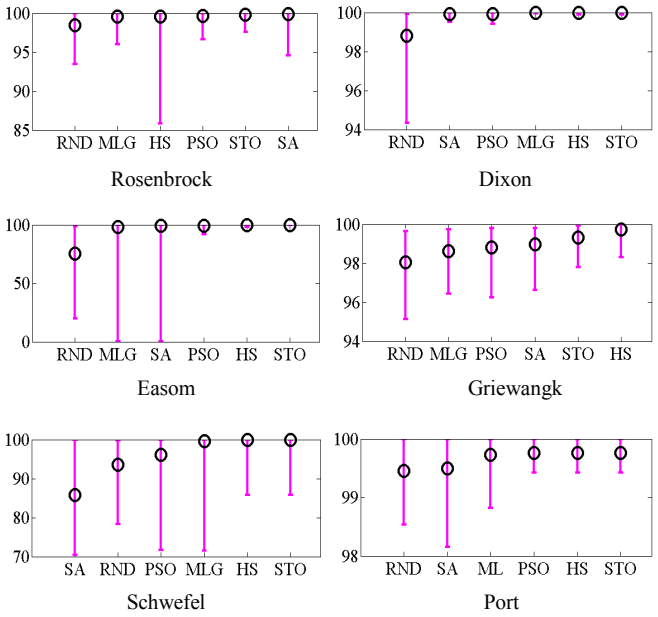

Figure 8. Function value reached after 500 function examinations as percentage of global maximum. Six test functions. Ordinate: median, 0.975 and 0.025 quantiles. The algorithms are sorted by their performance on the test functions.

approximate the steepest ascent. Each approximation of the gradient leads to two further evaluations. Thus, there is a trade-off between a focused search direction and the costs of computing the search direction. As expected, all other test functions showed better results using the gradient.

In another test, the inverse of the second derivation was used to control the step width of "thrown out" seeds like in the Gauss-Newton algorithm [3], but the result was less speed of convergence for most test functions, because two further function evaluations are needed for estimating the second deviation. With a synchronous parallel version all function evaluations for the derivations can be computed in parallel; thus in this case this option is appropriate to achieve a higher speed of convergence.

PSO was tested with 20, 30, 40, 50, 75 and 100 particles. For reaching 95% of the optimum, best results were achieved with 30 to 50 particles, except for Schwefel´s function (100 particles). After 500 function evaluations best results were achieved with 75 to 100 particles. This result is not surprising because the particles are initialized by random sampling and reaching the 95% mark was achieved by 57 to 95 function evaluations for all test functions except Easom and Schwefel.

Harmony search was tested with the best memory size of 5, 10 and 15. In most test functions B_{size}=5 showed best results.

In *Simulated Annealing* the exponential temper plan $T_0 c^t$ with c=0.8, initial temperature T_0=1 and maximum number of iterations of 1000 showed best results for most test functions.

Multi-Level Gradient Ascent was tested with several step widths and several maximum numbers of iterations for the vanilla Gradient Ascent. The results strongly depend on a right choice of the step width.

VI. CONCLUSION

After 500 function evaluations (Fig. 8) best results for Dixon´s, Easom´s and Schwefel´s function are achieved by Seed Throwing Optimization, the best result for Griewangk´s function is achieved by Harmony Search and the best results for Rosenbrock´s function by Simulated Annealing. Particle Swarm Optimization, Harmony Search and STO perform similarly for the port optimization application.

The test functions could be categorized into functions which are highly correlated and into functions where there is a low data-dependency in a local neighborhood. In functions with high correlation the gradient indicates the direction of the global maximum well. This is the case for Rosenbrock´s function, Dixon´s function and for the Port-function, whereas Easom´s, Griewangk´s and Schwefel´s function are more uncorrelated. We can conclude that Seed Throwing Optimization und Harmony Search performs best for both kinds of functions and are the best choices for the tested set of objective functions.

Another conclusion is that Particle Swarm Optimization, Multi-Level Gradient Ascent and Simulated Annealing perform better for the uncorrelated test functions than for the correlated ones.

In Simulated Annealing it was most difficult to find parameters which work well for all test functions.

Seed Throwing Optimization is suitable for synchronous parallelization but excels in asynchronous parallelization. Harmony Search and Particle Swarm Optimization are also well suitable for parallelization. Simulated Annealing and Gradient Ascent do not seem to be suitable for maximum performance in parallelization, since they contain strong dependencies in each iteration of the main loop.

The results (Fig. 7, Fig. 8) show that especially Seed Throwing Optimization and Harmony Search perform very well for a wide variety of functions. The solutions are highly reliable (0.025 quantile).

Moore´s law cannot be continued indefinitely. Hence parallel computing becomes more and more important. Many future computational applications in different domains comprise optimization problems. The asynchronously parallelized version of Seed Throwing Optimization is capable of solving optimization problems without insight of the objective function, it is easy to configure and very computationally efficient. Thus, future computational applications in many domains can benefit from using Seed Throwing Optimization.

ACKNOWLEDGMENT

The present study was conducted within the setting of the "Research training group 1126: Intelligent Surgery - Development of new computer-based methods for the future workspace in surgery" funded by the German Research Foundation.

REFERENCES

[1] C. Blum and A. Roli, "Metaheuristics in Combinatorial Optimization: Overview", ACM Comput. Surv. 35, 268-308 (2003).

[2] T. Weise, "Global Optimization Algorithms – Theory and Application", http://www.it-weise.de/projects/book.pdf [5-5- 2011].

[3] O. Weede, A. Kettler and H. Wörn, "Seed Throwing Optimization: A Probabilistic Technique for Multimodal Function Optimization," 2009 Computation World, pp. 515-519, 2009.

[4] Internet: http://www.intuitivesurgical.com/ [5-5-2011]

[5] Z.W. Geem, J.H. Kim and G.V. Loganathan, "A new heuristic optimization algorithm: Harmony search," Simulation 76, 60-68, 2001.

[6] J. Kennedy and R. Eberhart, "Particle Swarm Optimization", Proc. IEEE Int. Conf. on Neural Networks, 1995.

[7] S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi, "Optimization by Simulated Annealing", Science, New Series, Vol. 220, No. 4598, pp. 671-680, 1983.

[7] D. R. Greening, "Parallel Simulated Annealing Techniques", Physica D: Nonlinear Phenomena, Elsevier, pp.293-306, 1990

[8] J.F. Schutte, J.A. Reinbolt, B.J. Fregly, R.T. Haftka and A.D. George, "Parallel global optimization with the particle swarm algorithm", Int. J for Numerical Methods in Engineering, pp.61:2296-2315, 2004

[9] C. Jay and R. Hubbold, "Delayed Visual and Haptic Feedback in a Reciprocal Tapping Task", Proc. of the First Joint Eurohaptics Conf. and Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems, IEEE, pp. 655 – 656, 2005

[10] H. Pohlheim, "GEATbx: Example Functions 2 Parametric Optimization", Internet: http://www.geatbx.com/docu/fcnindex-01.html#P160_7291 [5-5-2011]

[11] G. Riley, "Network Simulation with ns-3", Spring Simulation Conference, April 12, 2010

[12] H. Wörn and O. Weede, "Optimizing the setup configuration for manual and robotic assisted minimally invasive surgery," in World Congress on Medical Physics and Biomedical Engineering 2009, Munich, Germany, 2009, pp. 55—58, 2009

 ISBN: 978-1-61208-154-0