Self-organized Invasive Parallel Optimization with Self-repairing Mechanism

Sanaz Mostaghim, Friederike Pfeiffer and Hartmut Schmeck

Institute AIFB
Karlsruhe Institute of Technology (KIT)
76128 Karlsruhe, Germany
{firstname.lastname}@kit.edu

Abstract: The parallelization of optimization algorithms is very beneficial when the function evaluations of optimization problems are time consuming. However, parallelization gets very complicated when we deal with a large number of parallel resources. In this paper, we present a framework called Self-organized Invasive Parallel Optimization (SIPO) in which the resources are self-organized. The optimization starts with a small number of resources which decide the number of further required resources on-demand. This means that more resources are stepwise added or eventually released from the platform. In this paper, we study an undesired effect in such a self-organized system and propose a self-repairing mechanism called Recovering-SIPO. These frameworks are tested on a series of multi-objective optimization problems.

1 Introduction

Inspired by invasive algorithms [Tei08], here we investigate a framework for parallel optimization. Invasive algorithms are proposed as a way to manage the control of parallel execution in multi-processor systems with hundreds of resources by giving the power of coordination and execution to the resources. This is proposed as a new self-organizing computing paradigm [Tei08]. In parallel optimization, coordination among the resources is more critical as the solutions obtained in different resources highly depend on each other. This leads to a high dependency between the data in the parallel resources, particularly if the parallelization model is aimed to divide the problem into smaller sub problems. In this paper, we investigate a new variant of our proposed framework Self-organized Invasive Parallel Optimization (SIPO) [MPS11]. In SIPO, the computation starts by giving the optimization problem to one available computing resource. According to the user preferences, the starting processor performs a rough optimization in the given partition and stores its results in a shared memory. Thereafter, if this partition passes a selection mechanism, the processor divides it into smaller ones. The smaller partitions are stored in a list which is accessible by other processors. As soon as a processing unit takes a job (partition) for optimization, it performs the same procedure described above and either assigns new jobs for other processors or stops the optimization process in its partition. In this way, the approximation of the optimal solutions, which are stored in the shared memory, gets more precise over the successive divisions of the tasks. From another point of view, the optimization starts from one computing resource and then successively occupies more resources

for computation until the computing resources cannot obtain any better solutions. The parallel resources are released one by one until there is no computing resource required for the optimization and in fact the system stops automatically. The selection mechanism for further divisions of a certain partition has a great impact on the quality of solutions. It may be the case that in the course of the selection mechanism some partitions get lost although they contain part of the optimal solutions.

In this paper, we focus on this issue in SIPO and propose a self-repairing method called Recovering-SIPO (R-SIPO). In R-SIPO, one of the resources detects the failing partitions and starts a new invasive parallel optimization only on these critical parts. Self-repairing methods are very often used in the area of software engineering e.g., [GSRU07]. However to our knowledge, the proposed self-repairing mechanism has not being studied in the area of parallel optimization. A recovering mechanism is introduced in [SMDT03] which deals with failures in the parameter space. This method is used for non-parallel algorithms. The main difference between SIPO and R-SIPO with the existing parallel optimization approaches (e.g., [TMO$^+$08]) is the varying number of resources over the time. While in traditional parallel approaches the parallel resources are active over the optimization time, in our framework, the resources are successively added to and deleted from the system, i. e. they are used on-demand.

The experiments on our case study show that we can significantly improve the results obtained by a parallel system like SIPO on multi-objective problems. In addition, we measure the speed-up factor of both frameworks SIPO and R-SIPO and compute an efficiency factor when comparing them with non-parallel algorithms and another parallel variant. The rest of this paper is organized as follows. We briefly explain SIPO in the next section and introduce R-SIPO in Section 3. In Section 4, we study SIPO and R-SIPO on our case study on multi-objective optimization and the experiments are investigated in Section 5. The paper is concluded in Section 6.

2 Self-Organized invasive parallel optimization (SIPO)

In SIPO, we consider a parallel platform of computing resources communicating through a shared memory. To start SIPO, the user is asked to give a rough estimation of the position of the desired solutions. This can be a partition or an interval, for instance $[0, 10]$ for an objective function. This initial partition is written in a global list located in the shared memory [MNS87]. This rough estimation is only used in the beginning of the algorithm and can even contain an infeasible area. Every processing unit which is able to perform a new job, repeatedly executes the following steps: (a) takes a partition from a global list of partitions in the shared memory, (b) optimizes inside the partition, (c) communicates the results to other resources through the shared memory, (d) evaluates the obtained results in its partition in order to decide for selecting the partition for further divisions, and (e) in case of further divisions, divides the partition into a number of smaller partitions and writes them on the global list. These steps are executed until there is no partition left in the global list of partitions. This global list is called problem-heap and is only used to store the partitions to be optimized. In addition, the resources communicate their obtained solutions to other resources by using a global archive which is also kept in the shared memory. This archive contains the best so-far-obtained solutions and is updated by the resources after

Algorithm 1: SIPO

repeat
  $I := GetPartition(problem\text{-}heap)$
  $A := GetGlobalArchive(A)$
  $a := Optimize(I, A)$
  Decide if I is kept for further divisions:
  if *($SelectInterval(I, A, a)$ ==TRUE)* **then**
    $List_I := Divide(I, NumofDivisions)$
    $problem\text{-}heap := UpdateProblemHeap(problem\text{-}heap, List_I)$
  end
  $A := UpdateGlobalArchive(A, a)$
until *I is empty*

each optimization step. In this way, the resources synchronize their results which is a very important aspect in parallel optimization. Algorithm 1 demonstrates the skeleton of SIPO which is executed by the resources. By the Functions $GetPartition(problem\text{-}heap)$ and $GetGlobalArchive(A)$, every resource downloads the global data such as the partition I to be optimized and the global archive A from the shared memory. $Optimize(I, A)$ runs an optimization algorithm which focuses on the given partition I according to the given global archive A. The solutions of the optimization are denoted as a. Before updating the global archive A (in $A := UpdateGlobalArchive(A, a)$), the Function $SelectInterval(I, A, a)$ decides if the partition I is kept for further divisions or not. For making this decision, the results a obtained in the partition are analyzed and compared with the stored results in the global archive A as in [MPS11]. I is kept for further divisions, if (i) a part of the global archive A is located within the interval and (ii) some improvements of the results a in terms of quality in respect to A are observed. If the result of $SelectInterval$ is true, the Function $Divide$ divides I into $NumofDivisions$ and write them in $List_I$. The $problem\text{-}heap$ is updated by the $List_I$ to make the new partitions accessible to the other processors. Thereby, the partition I is deleted from the problem-heap.

The partitions which do not survive the selection mechanism are not considered anymore for the rest of the computations. In fact, after a certain number of divisions, many intervals are not selected anymore and the number of required processors reduces until the results are not being improved and the algorithm stops automatically. This is a novelty of SIPO to automatically stop the optimization as it is usually a challenge to know when to stop a search algorithm [TLMP08]. Moreover, it utilizes the processors on-demand, i.e., if we do not obtain any better solutions, we do not waste resources. However, the selection mechanism has a great impact on the quality of solutions. It may be the case that in the course of the selection mechanism some partitions get lost although they contain a part of the optimal solutions. This will be discussed more into detail in the next section. SIPO is a decentralized approach, but from a central point of view, this invasive algorithm is like successive divisions of partitions through several iterations where each partition is assigned to a processing unit. Figure 1 illustrates an example of partitioning task in a two dimensional space with p indicating the problem-heap. Suppose, we start with 2 partitions ($|p| = 2$). These partitions are successively divided by $NumofDivisions = 2$ into 4 and 8 partitions. In the next division step, only 5 out of 8 partitions are further divided which

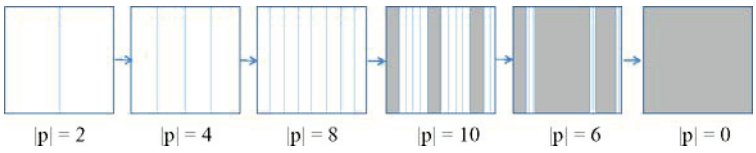

Abbildung 1: Schematic illustration of SIPO

results into $|p| = 10$ partitions. After one further step, there is no partition left for another division and the problem-heap is empty indicated by $|p| = 0$.

3 Self-repairing Mechanism

A major drawback of self-organization as explained above (in SIPO) is that the resources might neglect some partitions for further optimization due to the selection mechanism. This effect can be lessened by changing the parameters of SIPO, such as the significance levels or the selection mechanism in general [MPS11]. However, our concern here is to investigate a mechanism so that the resources evaluate their obtained results at the end of the entire optimization process and start a self-repairing mechanism if needed. A self-repairing mechanism can be performed in different ways. In this paper, we study a self-repairing mechanism called static recovering. The basic idea of static recovering is that at the end of SIPO, a recovering mechanism repairs the failures in the obtained results. Failures in our case are defined as gaps in the partitions.

For realizing the static recovering, the last processor working in SIPO starts the process once it has delivered his solutions to the shared memory. It must be aware that it was the last processor by examining if $|problem\text{-}heap|$ equals to 1 as illustrated in Algorithm 2. This is the case, if the processor does not divide its partition and there is no other partition left in the problem-heap. This processor scans the results stored in A for possible gaps and writes them as new partitions in the problem-heap. In this way, SIPO starts working again on the found gaps in a successive way. In other word, the recovering starts a new SIPO like a new impulse to the system. Algorithm 2 shows SIPO with the recovering mechanism. In this algorithm, we additionally assign an ACTIVE tag by $SetTag(I, problem\text{-}heap, ACTIVE)$ to the partitions which are taken by the resources. This tag is kept ACTIVE until the resource finishes its optimization process in that partition. The partition I is deleted by updating the problem-heap within $UpdateProblemHeap$. If the considered partition I was the last partition in the problem-heap so that $|problem\text{-}heap|$ == 1, the last processor has to find the failures, i.e. the gaps, in the global archive.

Finding Gaps: In Algorithm 2, the inputs to the Function $FindGaps$ are the global archive A before updating it with the current archive a (i.e., the old global archive) and the new global archive denoted as A'. This function examines first if there is any improvement in A' comparing to A. An improvement in the results indicates that recovering was required and could be executed again on the results to gain more solutions. The Function $FindGaps$ additionally calculates the distances of each solution in A' to its right and left neighbors and stores the results in $A_{Distances}$. The maximal distance $Distances_{Max}$ of $A_{Distances}$ is determined and the mean of $A_{Distances}$ with $Distances_{Max}$ is compared with the mean of $A_{Distances}$ without $Distances_{Max}$. If the percental change is greater than a given

Algorithm 2: Recovering-SIPO

repeat
  $I := GetPartition(problem\text{-}heap)$
  $SetTag(I, problem\text{-}heap, ACTIVE)$
  $A := GetGlobalArchive(A)$
  $a := Optimize(I, A)$
  if *(*$SelectInterval(I, A, a)$ *==TRUE)* **then** // I is selected for further divisions
    $List_I := Divide(I, NumofDivisions)$
    $problem\text{-}heap := UpdateProblemHeap(problem\text{-}heap, List_I)$
    $A := UpdateGlobalArchive(A, a)$
  else if *(*$|problem\text{-}heap|$ *== 1)* **then** // I was the last partition: start recovering
    $A' := UpdateGlobalArchive(A, a)$
    $G := FindGaps(A', A)$
    $A := A'$
    $problem\text{-}heap := UpdateProblemHeap(problem\text{-}heap, G)$
  end
until *I is empty*

threshold T, the corresponding partition G_0 is created, the distances corresponding to the solutions are deleted from $A_{Distances}$ and the same procedure is continued on the rest of the distances. The resulting gaps through the iterations are denoted as G_i. This loop is continued until the $Distances_{Max}$ reaches the threshold T. If no more gaps are found, the processor updates the problem-heap using the Function $UpdateProblemHeap$ by writing G for further optimization.

Discussion: The self-repairing mechanism is meant to improve the quality of the solutions by revisiting some already deleted partitions in the course of the selection mechanism. This is performed automatically and without involving any user. The advantage of our approach is that it considers the system as a whole and the results are obtained in a cooperative way by the resources. The possible failures denoted as gaps can occur at any time and along the execution of SIPO by any of the resources. The recovering process as suggested above is like a new impulse to the system and invasively involves new resources to work on the recovering process like in SIPO. The main advantage is that the recovering process automatically detects the failures, if any, and improves the results. However, by using this kind of recovering, there is no guarantee that all the failures are detected and recovered. This highly depends on the location of the optimal solutions. In some cases (as in our experiments) solutions are located on disconnected areas separated by gaps. The recovering process may require some time to differentiate between the real and failing gaps.

4 Case Study

SIPO is considered to be a general framework for self-organized parallel optimization. Here, we take multi-objective optimization as the case study like in [MPS11] and analyze the self-repairing mechanism. Typically a Multi-Objective Problem (MOP) involves several objectives to be optimized simultaneously: $\min_{\vec{x} \in S \subset \mathbb{R}^n} f_i(\vec{x}), \quad \text{for } i = 1 \ldots m$. The objective function is multi-dimensional $\vec{f} : \mathbb{R}^n \to \mathbb{R}^m$. We denote the image of S by $Z \subset \mathbb{R}^m$ and call it the objective space, the elements of Z are called objective vectors. Since we are dealing with MOPs, there is not generally one global optimum but a set of

Tabelle 1: Test functions

test	function	constraints
ZDT1	$g(x_2, \cdots, x_n) = 1 + 9(\sum_{i=2}^{n} x_i)/(n-1)$	$x_i \in [0,\ 1]$
	$h(f_1, g) = 1 - \sqrt{f_1/g}$	$n = 30$
	$f_1(\vec{x}) = x_1$	$i = 1, 2, \ldots, n$
	$f_2(\vec{x}) = g(x_2, \cdots, x_n) \cdot h(f_1, g)$	
ZDT2	$g(x_2, \cdots, x_n) = 1 + 9(\sum_{i=2}^{n} x_i)/(n-1)$	$x_i \in [0,\ 1]$
	$h(f_1, g) = 1 - (f_1/g)^2$	$n = 30$
	$f_1(\vec{x}) = x_1$	$i = 1, 2, \ldots, n$
	$f_2(\vec{x}) = g(x_2, \cdots, x_n) \cdot h(f_1, g)$	
ZDT3	$g(x_2, \cdots, x_n) = 1 + 9(\sum_{i=2}^{n} x_i)/(n-1)$	$x_i \in [0,\ 1]$
	$h(f_1, g) = 1 - \sqrt{f_1/g} - (f_1/g)\sin(10\pi f_1)$	$n = 30$
	$f_1(\vec{x}) = x_1$	$i = 1, 2, \ldots, n$
	$f_2(\vec{x}) = g(x_2, \cdots, x_n) \cdot h(f_1, g) + 1$	

so-called **Pareto optimal solutions**. A decision vector $\vec{x}_1 \in S$ is called **Pareto-optimal** if there is no other decision vector $\vec{x}_2 \in S$ that **dominates** it: $\vec{x}_1$ is said to dominate $\vec{x}_2$ if $\vec{x}_1$ is not worse than $\vec{x}_2$ in all of the objectives and it is strictly better than $\vec{x}_2$ in at least one objective. An objective vector is called Pareto-optimal if the corresponding decision vector is Pareto-optimal. Multi-objective optimization algorithms which properly fit into SIPO must be able to focus on a given partition either in parameter or objective space. In [MPS11], we demonstrated that it is relatively straightforward to deal with the focusing feature in the objective space. We assume that each partition is an interval in terms of one objective function. The focusing mechanism in [MPS11] is based on a multi-objective PSO algorithm called F-MOPSO.

5 Experiments

In the experiments, we analyze the self-repairing mechanism and its related parameters on our case study. We consider a parallel platform of homogeneous resources with many processors executing the same procedure as indicated by SIPO and Recovering-SIPO (R-SIPO). Furthermore, we assume that the communication overhead is negligible compared to optimization in a given interval. We select three different test problems containing convex, concave and disconnected fronts (included some gaps) denoted as ZDT1, 2 and 3 functions [DTLZ02] in Table 1. As in SIPO the quality of solutions gets more precise over the division process, the optimization algorithm in each interval must only find a rough approximation of solutions. Therefore, F-MOPSO is run for 400 evaluations in all the experiments. In R-SIPO, we decrease this value to 300 as the goal of recovering is to detect possible failures and do fine tuning. The maximal global archive size is kept to 100 by a clustering mechanism [MT03]. We select the standard value 0.4 for inertia weight. All the experiments are repeated for 30 different runs. The starting interval is selected as $[0, 1]$ along the first objective for all the test problems. We select the best parameter setting for SIPO from [MPS11] and employ it to R-SIPO in our experiments in this paper. The quality of solutions is computed by measuring the average number of function evaluations ($\#EV$) and the quality of solutions (HV) based on the hypervolume metric (smetric in [Zit99]) with $(3, 3)$ as the reference point.

Results: We first start with a statistical analysis on the threshold value T from Section 3. In order to analyze this factor which only affects the recovering process, we run SIPO

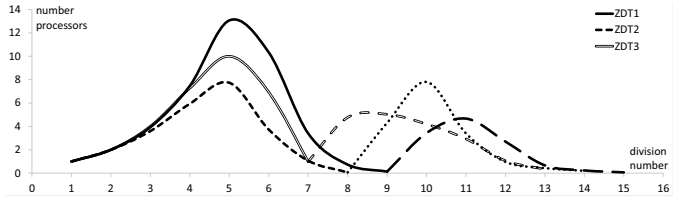

Abbildung 2: Average number of processors over successive divisions are shown for the three test problems.The continuing dashed lines indicate the recovering process.

first for 30 different runs and report the average hypervolume HV and its standard error SE_{HV} in the first part of Table 2. We execute only the recovering part of R-SIPO on the results of SIPO for different T values and report the extra required number of evaluations $\#Ev$ by the recovering and the overall quality of solutions in the second part of Table 2. The results show that the recovering has considerably improved the results and their corresponding standard error values particularly for ZDT2 and ZDT3. The low values for the standard errors for almost all the runs are remarkable. Comparing the results for different T indicates that it should be set to 0.04 for the convex problems and 0.06 for the concave whereas 0.05 namely the middle value in the experiments must be selected for the problems with gaps on the front. However, we notice that the selected range of 0.04 to 0.07 results in relatively good quality of solutions and the recovering mechanism is not very sensitive to this. In Table 2, $\#AMP$ is the average maximum number of processors required for SIPO whereas $\#MAP_D$ indicates the maximum of the average number of processors required in one division in SIPO and for R-SIPO, these values refer to the extra number of processors for only the recovering, respectively. Furthermore, the column indicated by $Recovering?$ illustrates the percentage of the runs which perform recovering. For all the SIPO runs, the value is 0 as SIPO does not perform recovering. For R-SIPO, we observe that a recovering mechanism was always required for the ZDT3 problem. This is because of the disconnected front of solutions as, at the beginning of the recovering, all the gaps are first considered as failures. In the case of ZDT1 and ZDT2, which both have connected fronts, R-SIPO is executed with a very high percentage of about 90 percent which demonstrates that we still are dealing with gaps along the connected front. Figure 2 illustrates the average maximum number of required processors $\#AMP$ before and after recovering. The number of resources at each division illustrates the parallel resources required at the same time. As expected, the recovering works like a new impulse for further invasive computations. The maximum number of required parallel processors is not as large as in SIPO, but it illustrates that the algorithms detect regions which require repairing.

Speed-up: In the following, we compute the speed-up of our invasive parallel approaches in different ways [JC09]. The first speed-up factor is considered as the ratio of the total execution time on a uniprocessor T_s to the total execution time of the parallel system T_p: $S_p = \frac{T_s}{T_p}$. For the computations, we assume that evaluation time of one solution takes one time unit (one second). We examine the required time T_s for the uniprocessor[1] to achieve

[1]We select NSGA-II algorithm [DPA02] for solving MOPs for the uniprocessor with population size 100, 200 generations, 0.9 for $P_{crossover}$ with $\eta_c = 15$ and 0.033 for $P_{mutation}$ with $\eta_m = 20$.

Tabelle 2: Results for SIPO and R-SIPO

	test function	T	Recovering?	HV	SE_{HV}	$\#Ev$	$SE_{\#Ev}$	$\#AMP$	MAP
SIPO									
	ZDT1	-	0	8.2259	0.0426	17668	917	42.1	13.1
	ZDT2	-	0	7.6480	0.1389	10556	809	25.1	7.7
	ZDT3	-	0	9.2372	0.0800	13524	463	32.2	10.0
R-SIPO									
	ZDT1	0.07	97	8.5124	0.0364	2549	271	8.0	3.7
		0.06	97	8.5486	0.0322	3680	390	11.5	5.6
		0.05	97	8.5616	0.0311	4864	481	15.2	6.2
		0.04	100	**8.5835**	0.0240	**5472**	410	17.1	7.8
	ZDT2	0.07	83	8.2587	0.0364	3776	602	11.8	4.6
		0.06	83	**8.3027**	0.0078	**3765**	516	11.8	4.7
		0.05	87	8.2772	0.0354	4117	536	12.9	4.9
		0.04	90	8.2297	0.0577	4299	515	13.4	5.1
	ZDT3	0.07	100	10.2087	0.0511	5717	368	17.9	4.7
		0.06	100	10.2358	0.0551	5888	322	18.4	5.4
		0.05	100	**10.2757**	0.0485	**5867**	343	18.3	5.0
		0.04	100	10.2050	0.0651	5483	268	17.1	4.9

Tabelle 3: Results of speed-up for SIPO and R-SIPO

	test function	T_p	T_s	HV_p	HV_s	$\#p$	S_p	E_p	Sx_p
SIPO									
	ZDT1	3780	5400	8.2259	7.7611	4.67	1.43	0.31	1.06
	ZDT2	3360	7400	7.6480	5.1768	3.14	2.20	0.70	1.48
	ZDT3	2940	3300	9.2372	9.0257	4.60	1.12	0.24	1.02
R-SIPO									
	ZDT1	5700	9300	8.5835	8.2929	3.57	1.63	0.46	1.04
	ZDT2	5280	16600	8.3027	6.7666	2.17	3.14	1.45	1.23
	ZDT3	4860	6700	10.2757	9.9123	3.88	1.38	0.36	1.04

the same quality of solutions as the parallel algorithms. With the selected parameters, T_p and T_s are measured as in Table 3 for SIPO and R-SIPO. The corresponding S_p values indicate that SIPO and R-SIPO have both larger speed-up values than one for all the test problems. This means that our parallel systems even as invasive models can achieve some speed-up values. The speed-up value for ZDT2 is remarkable and comparing SIPO and R-SIPO, we observe that recovering has significantly increased the speed-up for this problem.

Additionally, we compute the efficiency of the parallel algorithm based on these values. Efficiency E_p is measured as the fraction of time that a processor is effectively used [KGG94]. This is the ratio between the speed-up S_p and the number of processors $\#p$ used: $E_p = \frac{S_p}{\#p}$. The ideal value of E_p is one where usually it varies between zero and one. In our case, as we do not have a fixed number of parallel processors over time, we take the average of the average number of required processors over time as shown in Figure 2 and in Table 3. The efficiency values (E_p) illustrate that SIPO has very high efficiency when applied to ZDT2 problem. This value is even more than one after recovering as reported for R-SIPO and means that R-SIPO finds solutions which can not be found by the algorithm from [DPA02] on the uniprocessor. Overall, the efficiency values are improved by R-SIPO for all the test problems. This reveals that recovering has considerably contributed to improving the quality of the results.

Another speed-up factor for multi-objective optimization is called fixed-time-model [Gus90].

By this measurement, we assume that we have a time limit and compare the quality of the solutions obtained by the parallel system with the solutions of the uniprocessor. We take T_p as the time limit from above and measure the ratio called Sx_p as the fraction between the quality (here hypervolume) of solutions from the parallel system as reported in Table 2 and the quality of solutions from uniprocessor: $Sx_p = \frac{HV_p}{HV_s}$. We slightly changed the Sx_p from [Gus90] as we maximize the hypervolume and want to keep the definition of speed-up consistent over the paper. The calculated values of Sx_p in our experiments are reported in Table 3. By this measurement, we conclude that the results from SIPO and R-SIPO have better quality than the results from the uniprocessor. Comparing SIPO and R-SIPO, we observe that Sx_p values have been improved after the recovering for ZDT3 problem, where for ZDT1 and ZDT2, we have less improvements. One reason for this phenomenon is that we already obtained very good results for ZDT1 and ZDT2 with SIPO. R-SIPO has more contributed to the quality of the results for ZDT3 because this function contains several gaps along the optimal front and it is more likely that some good solutions are deleted by SIPO and then recovered by R-SIPO.

Comparison: For a more in-depth evaluation, we additionally compare the results of R-SIPO with another parallel alternative. We take the average of the average number of parallel processors over all divisions by R-SIPO (as illustrated in Figure 2). The average values are denoted as $\#p$ in Table 3. We divide the given interval into sub-intervals and run the processors in parallel for the same number of evaluations as in R-SIPO. In more details, we take $\#Ev$ from R-SIPO (23140, 14321 and 19391 for ZDT1 to ZDT3) and divide that by 4, 2 and 4. The number of iterations in each processor is computed as 288, 357 and 241 for ZDT1 to ZDT3 for 20 particles. We measure hypervolume values of 8.5802 (± 0.0111), 7.7836 (± 0.0282) and 10.3952 (± 0.0147) for the three problems respectively. R-SIPO obtains better quality of solutions for ZDT1 and ZDT2, whereas the obtained values for ZDT3 are surprisingly better than those from R-SIPO. This is due to the nature of the ZDT3 problem which contains a disconnected front of solutions and recovering spends more evaluations to repair the real gaps along the front. This indicates that it is difficult for R-SIPO to differentiate between real gaps and gaps caused by failures.

6 Conclusion and Future Work

In this paper, we study Self-organized Invasive Parallel Optimization (SIPO) for solving optimization problems using parallel platforms. Particularly, we investigate Recovering-SIPO (R-SIPO) which has a self-repairing mechanism to overcome the failures caused by the self-organization in SIPO. The studied failures here concern the gaps obtained in the course of selection mechanism in SIPO. SIPO starts from a partition given by the user with one resource. This resource performs a rough optimization in the partition and decides if the partition is further divided into smaller partitions or not. In the course of this selection mechanism, it can happen that some good partitions are not considered anymore in the optimization process. This causes undesired gaps in the solution space. We study a variant of recovering mechanism called static recovering as it starts working after the resources are done with the main steps of SIPO. The results of our experiments show that for the selected test problems the recovering improves the quality of the obtained solutions. This indicates that failures have indeed occurred by SIPO. In order to measure the power of recovering and to observe if R-SIPO can differentiate between real gaps and failures, we

intentionally select a test problem with gaps (namely ZDT3). The results on this problem illustrate that R-SIPO improves the quality but takes extra number of evaluations. This is observed by comparing R-SIPO with a parallel variant which could obtain equally good solutions with less number of evaluations than R-SIPO. We approved the result of the experiments by computing different speed-up measurements. In future, we intend to work on other variants of recovering methods and to minimize the number of evaluations by R-SIPO. Furthermore, as R-SIPO is a general framework, we will employ other optimization algorithms and investigate other test problems.

Literatur

[DPA02] K. Deb, A. Pratap und S. Agarwal. A Fast And Elitist Multi-objective Genetic Algorithm: NSGA-II. *IEEE Trans. on Evolutionary Computation*, 6(8), 2002.

[DTLZ02] K. Deb, L. Thiele, M. Laumanns und E. Zitzler. Scalable Multi-Objective Optimization Test Problems. In *Congress on Evolutionary Computation*, Seiten 825–830, 2002.

[GSRU07] D. Ghosh, R. Sharman, H. R. Rao und S. Upadhyaya. Self-healing systems - survey and synthesis. *Decision Support Systems*, 42:2164 – 2185, 2007.

[Gus90] J. L. Gustafson. Fixed Time, Tiered Memory, and Superlinear Speedup. In *Proceedings of the Fifth Distributed Memory Computing Conference*, Seiten 1255–1260, 1990.

[JC09] A. Lopez Jaimes und C. A. Coello Coello. Application of Parallel Plattforms and Models in Evolutionary Multi-Objective Optimization. In A. Lewis et al., Hrsg., *Biologically-inspired Optimisation Methods*, Seiten 23–49. Springer, 2009.

[KGG94] V. Kumar, G.K. Ananth Grama und A. Gupta. *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms*. Benjamin Cummings Publishing, 1994.

[MNS87] P. Moller-Nielsen und J. Staunstrup. Problem-heap: A paradigm for multiprocesor algorithms. *Parallel Computing*, 4(1):63 – 74, 1987.

[MPS11] S. Mostaghim, F. Pfeiffer und H. Schmeck. Self-organized Invasive Parallel Optimization. In *Proceedings of the Internantional Workshop on Bio-inspired Approaches for Distributed Computing*, Seiten 49–56. ACM, 2011.

[MT03] S. Mostaghim und J. Teich. Strategies for finding good local guides in multi-objective particle swarm optimization. In *Swarm Intelligence Symposium*, Seiten 26–33, 2003.

[SMDT03] O. Schütze, S. Mostaghim, M. Dellnitz und J. Teich. Covering Pareto Sets by Multilevel Evolutionary Subdivision Techniques. In *International Conference on Evolutionary Multi-Criterion Optimization*, Seiten 118–132, 2003.

[Tei08] J. Teich. Invasive Algorithms and Architectures. *it - Information Technology*, 50(5):300–310, 2008.

[TLMP08] H. Trautmann, U. Ligges, J. Mehnen und M. Preuss. A Convergence Criterion for Multiobjective Evolutionary Algorithms Based on Systematic Statistical Testing. In *Parallel Problem Solving from Nature*, Jgg. 5199, Seiten 825–836. Springer, 2008.

[TMO+08] E-G. Talbi, S. Mostaghim, T. Okabe, H. Ichibushi, G. Rudolph und C. A. Coello Coello. *Parallel Approaches for Multiobjective Optimization*, Seiten 349–372. Springer, 2008.

[Zit99] E. Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. Shaker, 1999.