Optimization Methods and Software
Vol. 00, No. 00, Month 200x, 1–28

Sprouting search - an algorithmic framework for asynchronous parallel unconstrained optimization

Árpád Bűrmen† and Tadej Tuma†

(.... released ....)

Direct search optimization algorithms are becoming an important alternative to well established gradient based methods. Due to the fact that a single cost function evaluation may take a substantial amount of time, optimization can be a lengthly process. In order to shorten the run time one often resorts to parallel algorithms. Asynchronous algorithms are particularly efficient since they have no synchronisation points. This paper is an attempt to establish a convergence theory for a class of such parallel direct search algorithms. The notion of a search direction generator (SDG) is introduced. An algorithmic framework for parallel distributed optimization methods based on SDGs is presented along with the corresponding convergence theory. The theory almost completely decouples the step-size control from the sufficient descent requirement, which is necessary for the finite termination of the algorithm's inner loop. The proposed framework has several attributes considered very favorable in loosely coupled parallel systems (e.g. clusters of workstations), such as fault tolerance and scalability. The framework is illustrated by optimizing a set of test problems on a cluster of workstations. In all tested cases a speedup was obtained that increased with the increasing number of workstations. Fault tolerance and scalability of the framework were also demonstrated by removing and adding workstations to the cluster while an optimization run was in progress.

Keywords: asynchronous parallel optimization; direct search; convergence analysis; distributed computing; cluster computing
2000 Mathematics Subject Classifications: 65K05; 65Y05; 68W15; 90C56

1 Introduction

n-dimensional unconstrained optimization problems of the form $\min_{x\in\mathbb{R}^n} f(\boldsymbol{x})$, $f : \mathbb{R}^n \to \mathbb{R}$ often involve solving local optimization problems (i.e. finding a local minimum of a function by starting the search from some given initial point $\boldsymbol{x}_0$).

There exist many algorithms for solving unconstrained local optimization problems when not only $f(\boldsymbol{x})$, but also the gradient $\nabla f(\boldsymbol{x})$ is available for every point $\boldsymbol{x} \in \mathbb{R}^n$. Unfortunately when it comes to real-world optimization problems, reliable gradient information is not available since the cost function value is the result of some simulation or even measurement.

† University of Ljubljana, Faculty of Electrical Engineering, Tržaška cesta 25, SI-1000 Ljubljana
E-mail: arpadb@fides.fe.uni-lj.si

For solving such problems direct search methods can be used since they don't require gradient information. The development of many of the today known direct search methods began in the 1960s. Several methods were developed albeit only a few of them came with a convergence theory. Especially notorious is the Nelder-Mead simplex algorithm [1], which performs very well in practise, and for which recently cases were discovered [2–4] implying that it doesn't necessarily converge even for well behaved functions. Due to the lack of mathematical analysis direct search methods were mostly ignored by the optimization community for a long time. Nevertheless the increasing interest in practical use of optimization brought new life to the field in the late 1980s with the analysis of multidirectional search (MDS) [4, 5] and later patters search (PS) [6]. The notion of pattern search was extended to grid-based search (GBS) [7], frame-based search (FBS) [8], global convergence framework for unconstrained derivative-free minimization (GCF) [9] and recently [10]. Several practical results were obtained from the aforementioned frameworks. PS provided a convergence theory for MDS, Hooke-Jeeves search (HJ) [11], and several other algorithms. From GBS an algorithm with finite termination on quadratics emerged [12]. One of the most interesting results is the convergent Nelder-Mead algorithm proposed in [13, 14] which draws its foundations from GBS. A very good overview of direct search methods can be found in [3,15,16].

Optimization problems can easily overwhelm state-of-the-art hardware since a single cost function evaluation can take a long time. This calls for efficient parallel optimization algorithms which are expected to accelerate the search by taking advantage of multiple processors working in parallel. Among other, two major approaches to parallelization can be found in literature. The first one divides the trial points among processors (see for instance [5, 17]). The second one divides the space in subspaces and assigns one such subspace to each processor [18–21]. Throughout the search processors exchange their best-yet values of $f(\boldsymbol{x})$, their position in the search space, stepsize information, and possibly other auxiliary data in order to guide the search to a common goal: a local minimizer of $f(\boldsymbol{x})$.

First attempts at parallel optimization produced synchronous algorithms. Such algorithms have one or more synchronisation points. When a processor reaches the synchronisation point in the algorithm, it stops and waits until other processors also reach it. Generally information exchange takes place at synchronsation points. Several approaches to parallel synchronous optimization can be found in literature. See for instance [5, 17, 22, 23]. The approaches in [18–21] are also originally targeted for synchronous execution.

Synchronous execution has a major downside which becomes obvious as soon as the time required to evaluate the cost function starts to vary. Some processors finish sooner than the others. Due to synchronisation points all of them must wait for the last one to finish. If the variations of the evaluation

time are large, the total idle time can be significant when compared to the total processing time spent by the processors. The result is a decrease in acceleration. The cost function evaluation time can vary especially for iterative algorithms such as solving nonlinear or differential equations (which is the case in most simulators). In case the parallel processors are not tightly coupled, the delay introduced by the communication channel between individual processors may also become significant when compared to the cost function evaluation time. At the bottom line this also results in processor idle time and the effect again, is reduced acceleration.

There is no way to overcome the delay introduced by the communication channel, except by using one that is faster and more responsive. However there can be something done about the variations of the evaluation time. By removing the synchronisation points from the algorithm an asynchronous parallel algorithm is obtained. Such algorithms inherently have no idle time caused by cost function evaluation time variations. The removal of synchronisation points brings along some difficulties. An asynchronous system is harder to manage than a synchronous one due to the fact that the information available to a processor may be outdated.

To our best knowledge the only asynchronous parallel direct search algorithm in the literature is the asynchronous parallel pattern search (APPS) [24]. The algorithm is derived from PS. It assigns one of the search directions to every processor. As soon as a processor finds a better point in its direction (internal success) it broadcasts it to all other processors and optionally increases the stepsize. If it fails to find a better point the stepsize is decreased (failure). When a broadcast is received, a processor checks if the received point is better than its current best point. If it is, the received point and stepsize replace the current best point and stepsize (external success).

The convergence proof for APPS can be found in [25] and some additional explanations in [26]. The notation is similar to the one found in [27] for tracking iterates across processors in asynchronous parallel iterative algorithms. PS doesn't impose a sufficient descent condition. It requires only simple descent. Therefore the iterates must lie on a rational lattice in order to ensure convergence. This is achieved by imposing rationality requirements on search directions and stepsize change factors. The two requirements are sufficient for convergence to a stationary point of $f(\boldsymbol{x})$ for the sequential (and synchronous) PS. In case of the asynchronous algorithm more is required. Since every processor searches in its own direction one must ensure that a subset of processors, whose search directions build a positive spanning set [28], has a common accumulation point where the stepsize approaches zero. In APPS this is achieved by imposing an additional requirement on the stepsize change parameter. When the step is changed after an internal success the resulting stepsize parameter must be bounded by a lower and an upper bound. After

a lengthly proof one obtains that the stepsize goes to zero and that the processors share a common accumulation point. Consequently by means of same reasoning as in PS it follows that the accumulation point is also a stationary point of $f(\boldsymbol{x})$.

APPS has several disadvantages. First of all the stepsize parameter is common for all search directions. If the search in one direction requires short steps, search in all other directions also proceeds with short steps. Secondly a common accumulation point for all processors is guaranteed by limiting the stepsize after a successfull step. The stepsize limitation eventually leads to prolonged periods of stepsize decrease during which the best point in the system propagates across all processors. Some means for decoupling the stepsize control from the mechanism for providing a common accumulation point would be beneficiary and would make the algorithm and its convergence theory simpler. This would be of great benefit for defining new asynchronous parallel optimization algorithms based on well established sequential or synchronous parallel algorithms like the convergent Nelder-Mead simplex algorithm [13,14] or the parallel variable transformation [21].

The remainder of this paper is organized as follows. First some basic information regarding positive spanning sets, positive bases, and limit points of sequences of direction sets are provided. Next the notion of a search direction generator and its feasibility is introduced, illustrated by examples. The algorithmic framework is presented along with a convergence theory. The conditions assuring the convergence of algorithms conforming to the presented framework are discussed, followed by the framework's possibilities in the field of parallel asynchronous direct search. Special emphasis is given to the algorithm's run-time scalability and fault tolerance. Finally results obtained by a cluster of workstations running an algorithm conforming to the aforementioned framework are provided and discussed. The paper concludes with a summary of its main results.

Notation. $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, and $\mathbb{N}$ denote the sets of real, rational, integer, and natural numbers, respectively. Similarly $\mathbb{R}_+$, $\mathbb{Q}_+$ denote nonnegative real and rational numbers. Let $|\mathcal{A}|$ denote the number of elements in set $\mathcal{A}$, $\|\boldsymbol{x}\|$ the euclidean norm of $\boldsymbol{x}$, $\boldsymbol{x}^{\mathrm{T}}\boldsymbol{y}$ the scalar product of $\boldsymbol{x}$ and $\boldsymbol{y}$, and $\mathcal{L}_f(\boldsymbol{x}) = \{\boldsymbol{y} : f(\boldsymbol{y}) \leq f(\boldsymbol{x})\}$ a level set of $f(\boldsymbol{x})$. With $\gamma\mathcal{A}$ we denote the set whose members are members of set $\mathcal{A}$ scaled by γ. Similarly $-\mathcal{A}$ denotes the set whose members are members of set $\mathcal{A}$ multiplied by -1. In fact whenever an arithmetic operation $x \odot \mathcal{A}$ takes place with x being a scalar (a vector) and $\mathcal{A}$ being a set of scalars (vectors), the result is a set of all scalars (vectors) $x \odot a$ where a is a member of $\mathcal{A}$.

In the following sections we shall assume that the cost function $f(\boldsymbol{x})$ is continuously differentiable. A direction is any nonzero vector. We denote the dimensionality of the search space by n. Let $\mathcal{U}_{\mathrm{S}}^n$ denote the set of all possible

subsets of $\mathbb{R}^n$.

2 Generating the sequence of directions

The approach proposed herein differs from the one developed by Lucidi and Sciandrone [9] in the sense that directions are generated one at a time. The use of a search direction generator (SDG) broadens the scope of the framework. As it will become evident at a later point in the paper, SDGs can be capable of adapting to the cost function behaviour. The set of restrictions imposed on a feasible SDG serves two purposes. First of all to make possible the generation of an infinite sequence of candidate stationary points, and secondly to ensure that a positive spanning set of steps is examined around every limit point of the search.

2.1 *Positive spanning sets and search direction generators*

In this section some basic information regarding positive spanning sets is provided to the reader. Then the notion of a search direction generator is introduced and discussed.

Definition 2.1 The positive span of a set of vectors $\mathcal{D} = \{\boldsymbol{d}^1, \boldsymbol{d}^2, ..., \boldsymbol{d}^m\} \subset \mathbb{R}^n$ is a set defined as $span_+\mathcal{D} = \{\boldsymbol{x} : \boldsymbol{x} = \sum_{i=1}^m \alpha_i \boldsymbol{d}^i, \alpha_i \geq 0\}$.

Definition 2.2 $\mathcal{B} = \{\boldsymbol{b}^1, \boldsymbol{b}^2, ..., \boldsymbol{b}^m\}$ is a positive basis for $\mathbb{R}^n$ if it positively spans $\mathbb{R}^n$ and has no proper subset that positively spans $\mathbb{R}^n$.

A positive basis has at least $n+1$ members and no more than $2n$ members. Refer to [28] for a detailed explanation of positive bases and their properties. Positive spanning sets play an important role in convergence analysis of many direct search methods. The reason for this is the following lemma.

LEMMA 2.3 *Suppose that $\mathbb{R}^n$ is in the positive span of $\mathcal{P}$ and $\boldsymbol{x} \in \mathbb{R}^n$. Then the following deduction can be made*

$$\boldsymbol{x}^{\mathrm{T}}\boldsymbol{p} \geq 0, \forall \boldsymbol{p} \in \mathcal{P} \quad \Rightarrow \quad \boldsymbol{x} = 0. \tag{1}$$

Proof Since $\mathcal{P}$ positively spans $\mathbb{R}^n$, $-\boldsymbol{x}$ can be expressed as $-\boldsymbol{x} = \sum_{i=1}^m \alpha_i \boldsymbol{p}^i, \alpha_i \geq 0, \forall i$. Now we can write $-x^{\mathrm{T}}x = \sum_{i=1}^m \alpha_i \boldsymbol{x}^{\mathrm{T}}\boldsymbol{p}^i \geq 0$. But $-\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x} \leq 0$ must also hold. Both can be true only if $\boldsymbol{x} = 0$. □

Using the property described in lemma 2.3 direct search algorithms detect that $\boldsymbol{x}$ is a stationary point of a continuously differentiable function by evaluating the function on rays originating from $\boldsymbol{x}$ in directions defined by members of $\mathcal{B}$.

Let $\{\mathcal{P}\}$ denote a sequence of sets that positively span $\mathbb{R}^n$. Note that not necessarily all members of the sequence comprise the same number of directions.

Definition 2.4 $\mathcal{P}_\infty = \{\boldsymbol{p}_\infty^1, \boldsymbol{p}_\infty^2, ..., \boldsymbol{p}_\infty^m\}$ is a limit point of sequence $\{\mathcal{P}_k\}_{k=1}^\infty$ if and only if for every $\epsilon > 0$ there exist infinitely many $\mathcal{P} \in \{\mathcal{P}_k\}_{k=1}^\infty$, $|\mathcal{P}_\infty| = |\mathcal{P}| = m$ for which one can find $\{\boldsymbol{q}^1, \boldsymbol{q}^2, ..., \boldsymbol{q}^m\} = \mathcal{P}$ such that

$$\|\boldsymbol{p}_\infty^i - \boldsymbol{q}^i\| \leq \epsilon, \quad i = 1, 2, ..., m. \tag{2}$$

Definition 2.5 A sequence of positive spanning sets $\{\mathcal{P}_k\}_{k=1}^\infty$ is positively degenerate if it has some limit point $\mathcal{P}_\infty$ whose positive span does not include $\mathbb{R}^n$.

Lemma 2.6 *Suppose that* $\mathcal{P} = \{\boldsymbol{p}^1, \boldsymbol{p}^2, ..., \boldsymbol{p}^m\}$ *positively spans* $\mathbb{R}^n$. *Then* $\mathcal{Q} = \{\boldsymbol{q}^1, \boldsymbol{q}^2, ..., \boldsymbol{q}^r\}$ *positively spans* $\mathbb{R}^n$ *if every* $p \in \mathcal{P}$ *can be expressed as a nonnegative linear combination of members of* $\mathcal{Q}$.

Proof Since $\mathcal{P}$ positively spans $\mathbb{R}^n$, any $\boldsymbol{x} \in \mathbb{R}^n$ can be expressed as

$$\boldsymbol{x} = \sum_{j=1}^m \alpha_j \boldsymbol{p}^j, \quad \boldsymbol{p}^j = \sum_{i=1}^r \beta_{i,j} \boldsymbol{q}^i.$$

Now we can write

$$\boldsymbol{x} = \sum_{j=1}^m \alpha_j \sum_{i=1}^r \beta_{i,j} \boldsymbol{q}^i = \sum_{i=1}^r (\sum_{j=1}^m \beta_{i,j} \alpha_j) \boldsymbol{q}^i.$$

Due to $\alpha_j \geq 0$ and $\beta_{i,j} \geq 0$ it directly follows $\sum_{j=1}^m \beta_{i,j} \alpha_j \geq 0$. □

Lemma 2.6 implies that one can check whether $\mathcal{P}$ is a positive spanning set by simply solving m nonnegative least square (NNLS) problems of the form $\mathbf{V}\boldsymbol{y} = \boldsymbol{p}$ (one for every $\boldsymbol{p} \in \mathcal{P}$) where the columns of $\mathbf{V}$ are members of $\mathcal{Q}$. If a solution with $\|\mathbf{V}\boldsymbol{y} - \boldsymbol{p}\| = 0$ for all m NNLS problems is obtained $\mathcal{Q}$ positively spans $\mathbb{R}^n$.

Torczon [6] and later Lewis and Torczon [29] prooved the following lemma.

Lemma 2.7 *For any set* $\mathcal{P}$ *positively spanning* $\mathbb{R}^n$ *there exists* $\epsilon > 0$ *such that for any* $\boldsymbol{x} \in \mathbb{R}^n$, $\|\boldsymbol{x}\| = 1$ *there exists* $p \in \mathcal{P}$ *with* $\boldsymbol{p}^{\mathrm{T}} \boldsymbol{x} \geq \epsilon \|\boldsymbol{p}\|$.

Lemma 2.7 guarantees that for a given positive spanning set and any vector $\boldsymbol{x}$ there exists a member of the positive spanning set $\boldsymbol{p}$ such that the angle between $\boldsymbol{p}$ and $\boldsymbol{x}$ remains uniformly bounded away from $\pi/2$. This fact guarantees (if the step is sufficiently small) a fraction of steepest descent along some

member of the positive spanning set for continuously differentiable functions (see [6]).

Since the zero vector can be expressed as a nontrivial positive combination of members of $\mathcal{P}$ (add the positive combination for $\boldsymbol{x} \neq 0$ to the positive combination for $-\boldsymbol{x}$) there exist infinitely many ways to express some $\boldsymbol{x} \in \mathbb{R}^n$ as a positive combination of members of $\mathcal{P}$. The following lemma deals with the existence of a positive combination whose coefficients remain bounded with regard to the norm of $\boldsymbol{x}$.

LEMMA 2.8 *Suppose that $\mathcal{B}$ positively spans $\mathbb{R}^n$. Then for any $\boldsymbol{x} \in \mathbb{R}^n$ there exists a positive combination of members of $\mathcal{B}$ such that $\boldsymbol{x} = \sum_{i=1}^{|\mathcal{B}|} \alpha_i \boldsymbol{b}^i$ and $\sum_{i=1}^{|\mathcal{B}|} \alpha_i < C\|\boldsymbol{x}\|$ where C is a positive constant.*

Proof For every $\boldsymbol{x} \in \mathbb{R}^n$ and every positive spanning set $\mathcal{B}$ there exists

$$\boldsymbol{q} = \arg\max_{\boldsymbol{b} \in \mathcal{B}} \boldsymbol{x}^{\mathrm{T}} \boldsymbol{b} / (\|\boldsymbol{b}\| \|\boldsymbol{x}\|), \quad \boldsymbol{x}^{\mathrm{T}} \boldsymbol{q} > 0.$$

There exists such $\beta \geq 0$ that $\boldsymbol{x}$ can be decomposed as $\boldsymbol{x} = \beta \boldsymbol{q} + \boldsymbol{r}$ and $\boldsymbol{r}^{\mathrm{T}} \boldsymbol{q} = 0$. Lemma 2.7 assures us that there exists $0 \leq \epsilon = \cos\theta < 1$ such that $\boldsymbol{q}^{\mathrm{T}} \boldsymbol{x} \geq \epsilon \|\boldsymbol{q}\| \|\boldsymbol{x}\|$ where ϵ depends only on the choice of $\mathcal{B}$. The following estimate can be derived

$$\|\boldsymbol{r}\|^2 \leq \|\boldsymbol{x}\|^2 (1 - \epsilon^2) = \|\boldsymbol{x}\|^2 \sin^2\theta. \tag{3}$$

By starting with $\boldsymbol{x}$ and sequentially reapplying the above mentioned decomposition procedure to $\boldsymbol{r}$ a series for $\boldsymbol{x}$ is obtained

$$\boldsymbol{x} = \sum_{i=0}^{\infty} \beta_i \boldsymbol{q}_i, \quad \beta_i > 0, \ \ \boldsymbol{q}_i \in \mathcal{B}. \tag{4}$$

$\beta_0 \|\boldsymbol{q}_0\| \leq \|\boldsymbol{x}\|$, $\beta_i \|\boldsymbol{q}_i\| \leq \|\boldsymbol{r}_{i-1}\| \leq \|\boldsymbol{x}\| \sin^i\theta$ (for $i > 0$), and (3) yields the following estimate from (4)

$$\|\sum_{i=0}^{\infty} \beta_i \boldsymbol{q}_i\| \leq \sum_{i=0}^{\infty} \beta_i \|\boldsymbol{q}_i\| \leq \|\boldsymbol{x}\| \sum_{i=0}^{\infty} \sin^i\theta = \|\boldsymbol{x}\| (1 - \sin\theta)^{-1}. \tag{5}$$

On the other hand

$$\|\boldsymbol{b}_{\min}\| \sum_{i=0}^{|\mathcal{B}|} \alpha_i = \|\boldsymbol{b}^{\min}\| \sum_{i=0}^{\infty} \beta_i \leq \sum_{i=0}^{\infty} \beta_i \|\boldsymbol{q}_i\| \tag{6}$$

where $\boldsymbol{b}^{\min} = \arg\min_{\boldsymbol{b}\in\mathcal{B}} \|\boldsymbol{b}\|$. Now by joining (5) and (6) it follows that

$$\sum_{i=1}^{|\mathcal{B}|} \alpha_i = \sum_{i=1}^{\infty} \beta_i \leq \|\boldsymbol{b}^{\min}\|^{-1}(1-\sin\theta)^{-1}\|\boldsymbol{x}\|.$$

□

Let $\boldsymbol{b}^{\min}$ and $\boldsymbol{b}^{\max}$ denote $\arg\min_{\boldsymbol{b}\in\mathcal{B}} \|\boldsymbol{b}\|$ and $\arg\max_{\boldsymbol{b}\in\mathcal{B}} \|\boldsymbol{b}\|$ respectively. $\lambda > 0$ is a lower and Λ an upper bound on $\|\boldsymbol{p}\|$ for all $\boldsymbol{p} \in \mathcal{P}$.

LEMMA 2.9 *Let $\mathcal{B}$ denote a set that positively spans $\mathbb{R}^n$. Suppose that any $\boldsymbol{b} \in \mathcal{B}$ can be expressed in the NNLS manner as $\boldsymbol{b} = \mathbf{V}\boldsymbol{y}$ where $\|\boldsymbol{y}\| \leq C\|\boldsymbol{b}^{\max}\|/\lambda$, $C \geq 1$, and the columns of $\mathbf{V}$ are members of $\mathcal{P}$. Then there exists $0 < \epsilon < 1$ such that for any $\boldsymbol{x} \in \mathbb{R}^n, \|\boldsymbol{x}\| = 1$ some $\boldsymbol{p} \in \mathcal{P}$ can be found with $\boldsymbol{x}^{\mathrm{T}}\boldsymbol{p} \geq \epsilon\|\boldsymbol{p}\|$.*

Proof Due to lemma 2.6 $\mathcal{P}$ positively spans $\mathbb{R}^n$ and $\boldsymbol{x}$ can be expressed as

$$\boldsymbol{x} = \sum_{i=1}^{|\mathcal{P}|}\Big(\sum_{j=1}^{|\mathcal{B}|} \beta_{i,j}\alpha_j\Big)\boldsymbol{p}^i.$$

The norm of $\boldsymbol{x}$ can be written as

$$\|\boldsymbol{x}\|^2 = \boldsymbol{x}^{\mathrm{T}}\boldsymbol{x} = \sum_{j=1}^{|\mathcal{B}|}\Big(\sum_{i=1}^{|\mathcal{P}|} \beta_{i,j}\alpha_j\Big)\boldsymbol{x}^{\mathrm{T}}\boldsymbol{p}^i \leq \max_{\boldsymbol{p}\in\mathcal{P}} \boldsymbol{x}^{\mathrm{T}}\boldsymbol{p} \sum_{j=1}^{|\mathcal{B}|}\Big(\sum_{i=1}^{|\mathcal{P}|} \beta_{i,j}\alpha_j\Big).$$

$\|\boldsymbol{y}\| \leq C\|\boldsymbol{b}^{\max}\|/\lambda$ implies $\beta_{i,j} \leq C\|\boldsymbol{b}^{\max}\|/\lambda$. Then from Lemma 2.8 it follows

$$\begin{aligned} 1 &\leq C|\mathcal{P}|\|\boldsymbol{b}^{\max}\|\lambda^{-1} \max_{\boldsymbol{p}\in\mathcal{P}} \boldsymbol{x}^{\mathrm{T}}\boldsymbol{p} \sum_{j=1}^{|\mathcal{B}|} \alpha_j \\ &\leq C|\mathcal{P}|\|\boldsymbol{b}^{\max}\|\|\boldsymbol{b}^{\min}\|^{-1}(1-\sin\theta)^{-1}\lambda^{-1} \max_{\boldsymbol{p}\in\mathcal{P}} \boldsymbol{x}^{\mathrm{T}}\boldsymbol{p} \end{aligned}$$

where θ is defined as in the proof of Lemma 2.8. Finally $\boldsymbol{x}^{\mathrm{T}}\boldsymbol{p}/\Lambda \leq \boldsymbol{x}^{\mathrm{T}}\boldsymbol{p}/\|\boldsymbol{p}\|$ and therefore

$$\max_{\boldsymbol{p}\in\mathcal{P}} \frac{\boldsymbol{x}^{\mathrm{T}}\boldsymbol{p}}{\|\boldsymbol{p}\|} \geq \frac{\|\boldsymbol{b}^{\min}\|\lambda(1-\sin\theta)}{C\|\boldsymbol{b}^{\max}\|\Lambda|\mathcal{P}|}$$

□

Definition 2.10 A search direction generator (SDG) is a transformation:

$$G : (\boldsymbol{x}^{\mathrm{o}}, \mathcal{S}) \mapsto \boldsymbol{p}, \quad \mathcal{S} \in \mathcal{U}_{\mathrm{S}}^{n}, \;\; \boldsymbol{x}^{\mathrm{o}}, \boldsymbol{p} \in \mathbb{R}^{n}.$$

Definition 2.11 Suppose $\boldsymbol{x}^{\mathrm{o}} \in \mathbb{R}^{n}$, $\mathcal{S}^{0} \in \mathcal{U}_{\mathrm{S}}^{n}$, and $\mathcal{S}^{i}$ is obtained by iterating

$$\begin{aligned} \boldsymbol{p}^{i} &= G(\boldsymbol{x}^{\mathrm{o}}, \mathcal{S}^{i}) \\ \mathcal{S}^{i} \cup \{\boldsymbol{x}^{\mathrm{o}} + \boldsymbol{p}^{i}\} &\subseteq \mathcal{S}^{i+1}, \quad i = 0, 1, 2, ... \end{aligned} \tag{7}$$

Let $\mathcal{P}^{i}$ denote the set obtained from $\mathcal{S}^{i}$ with $\mathcal{P}^{i} = \mathcal{S}^{i} - \boldsymbol{x}^{\mathrm{o}}$. The search direction generator G is feasible for n-dimensional search if for any $\mathcal{S}^{0} \in \mathcal{U}_{\mathrm{S}}^{n}$ and any $\boldsymbol{x}^{\mathrm{o}} \in \mathbb{R}^{n}$ there exists such $n < N < \infty$ and $\mathcal{B} \subseteq \mathcal{P}^{N}$ that $\mathcal{B}$ has the following properties

$$\begin{aligned} \forall \boldsymbol{b} \in \mathcal{B} : \quad & \lambda \leq \|\boldsymbol{b}\| \leq \Lambda, \\ \forall \boldsymbol{q} \in \mathbb{R}^{n}, \|\boldsymbol{q}\| = 1, \exists \boldsymbol{b} \in \mathcal{B} : \quad & \boldsymbol{q}^{\mathrm{T}} \boldsymbol{b} \geq \|\boldsymbol{b}\| \cos\theta. \end{aligned} \tag{8}$$

The requirement $N > n$ arises from the fact that a minimal positive basis for $\mathbb{R}^{n}$ consists of $n + 1$ directions. The scalars λ, Λ, and θ are positive real constants satisfying $0 < \lambda \leq \Lambda$, and $0 \leq \theta < \pi/2$. Let $\mathcal{D}$ denote the set obtained by scaling the members of $\mathcal{B}$ with $1/\Lambda$.

Lemma 2.12 *Assume a sequence of sets $\{\mathcal{S}_{k}^{0}\}_{k=1}^{\infty}$ from $\mathcal{U}_{\mathrm{S}}^{n}$, a sequence of feasible SDGs $\{G_{k}\}_{k=1}^{\infty}$, and sequences of positive numbers $\{\lambda_{k}\}_{k=1}^{\infty}$ and $\{\Lambda_{k}\}_{k=1}^{\infty}$ ($0 < \lambda_{k} \leq \Lambda_{k}$). D_{k} is generated by applying the iteration (7) (starting with $\mathcal{S}_{k}^{0}$) using G_{k}, λ_{k}, and Λ_{k} until some $\mathcal{B}_{k}$ satisfying conditions in (8) is found. Suppose that $\lambda_{k}/\Lambda_{k} \geq \gamma$, $0 < \gamma \leq 1$. Then the resulting sequence $\{\mathcal{D}_{k}\}_{k=1}^{\infty}$ has at least one limit point $\mathcal{D}_{\infty}$ and every limit point positively spans $\mathbb{R}^{n}$.*

Proof Due to the first condition in (8) any direction from any $\mathcal{D}_{k}$ remains bounded ($\gamma \leq \|\boldsymbol{p}/\Lambda_{k}\| \leq 1$). Therefore the sequence $\{\mathcal{D}_{k}\}_{k=1}^{\infty}$ admits at least one limit point $\mathcal{D}_{\infty}$ and $\mathbf{0} \notin \mathcal{D}_{\infty}$. The second requirement in (8) ensures, that no matter how a nonzero vector $\boldsymbol{q} \in \mathbb{R}^{n}$ is chosen there exists at least one $\mathbf{p} \in \mathcal{D}_{k}$ such that the angle between q and p is bounded uniformly away from $\pi/2$. By means of an argument similar to the one in the proof of Lemma 2.8 one can come to the conclusion that $\mathcal{D}_{k}$ positively spans $\mathbb{R}^{n}$. Since the second requirement must hold for all $\mathcal{D}_{k}$, it must also hold for $\mathcal{D}_{\infty}$. Finally it follows that $\mathcal{D}_{\infty}$ positively spans $\mathbb{R}^{n}$. □

2.2 *An example of a search direction generator*

It can be of great advantage if the SDG is capable of detecting whether the set $\mathcal{P}_i$ has some subset that is a positive basis for $\mathbb{R}^n$ and satisfies (8). In such case it is supposed to produce a zero direction vector $\boldsymbol{p} = \mathbf{0}$ signaling the optimization algorithm that a positive basis has been found. It is not of vital importance to the convergence properties of the algorithm that the positive basis is detected as soon as it appears in $\mathcal{P}_i$. Its detection may occur later as long as the value of i when this happens has an upper bound. The fact that the SDG itself decides whether a positive basis has been found greatly simplifies the asynchronous algorithm and enables us to incorporate the information on steps that failed to produce (sufficient) descent, received from other workers into the search strategy.

Algorithm 1 A SDG based on NLLS.

if $|\mathcal{S}| = 0$ **then**
  return first member of $\mathcal{B}$;
end
Let columns of $\mathbf{V}$ be vectors $\boldsymbol{x} - \boldsymbol{x}^{\mathrm{o}}$
  where $\boldsymbol{x} \in \mathcal{S}$, $f(\boldsymbol{x}) \geq f(\boldsymbol{x}^{\mathrm{o}}) - h$, and $\lambda \leq \|\boldsymbol{x} - \boldsymbol{x}^{\mathrm{o}}\| \leq \Lambda$.
for $i = 1, 2, ..., |\mathcal{B}|$ **do**
  obtain $\boldsymbol{y}$ by solving the NLLS problem $\mathbf{V}\boldsymbol{y} = \boldsymbol{b}^i$;
  if $\|\boldsymbol{y}\| > C\|\boldsymbol{b}^{\max}\|\lambda^{-1} \quad \vee \quad \|\mathbf{V}\boldsymbol{y} - \boldsymbol{b}^i\| > D$ **then**
    return $\boldsymbol{b}^i$;
  end
end
return zero vector;

Algorithm *1* represents a possible way of choosing the search direction. Let $\mathbf{V}$ denote a matrix whose columns are members of $\mathcal{P}$. C is a positive constant. If $C \geq 1$ the SDG generates a positive spanning set in at most $|\mathcal{B}|$ iterations of the form (7). Note that iteration (7) allows for arbitrary points to be added to set $\mathcal{S}$. These points mean that some regions of the search space are already examined and the corresponding trial steps need not be considered. So in practise one can expect that less than $|\mathcal{B}|$ iterations are needed to produce a positive spanning set.

Provided that $\|\boldsymbol{b}^{\max}\|/\|\boldsymbol{b}^{\min}\|$, Λ/λ, and $|\mathcal{P}|$ remain bound from above and the sequence $\{\mathcal{B}_i\}_{i=1}^{\infty}$ is not positively degenerate, the corresponding sequence of positive spanning sets generated by the SDG cannot be positively degenerate. This is guaranteed by Lemma 2.9. Together with the enforced inequality $\lambda \leq \|\boldsymbol{p}\| \leq \Lambda$ it follows that this SDG is feasible.

The number of trial steps contributed by the SDG before $\boldsymbol{p} = \mathbf{0}$ is returned

is bound from above by $|\mathcal{B}|$. The time needed to complete $|\mathcal{B}|$ iterations of the form (7) is finite. In a finite time a finite number of external search processes can contribute only a finite number of points to set $\mathcal{S}$ and therewith a finite number of trial steps to set $\mathcal{P}$.

3 The sprouting search algorithm

Sprouting search (algorithm *2*) looks for a stationary point by examining trial points and repeatedly moving the origin of the search $\boldsymbol{x}^{\mathrm{o}}$ to a better point (a point with a lower cost function value). The examined trial points lie around the current origin. When (sufficient) descent is obtained, the origin of the search changes. If the algorithm is parallelized every worker can have its own search origin. By connecting the origins with the points examined around them we obtain a picture resembling an evolving plant (thus the name sprouting search).

The fact that the cost function value decreases with every move of the origin is not sufficient to obtain convergence to a stationary point, except in some special cases where the set of examined points lies on a scaled rational lattice (PS, [6]). By taking into account certain restrictions the structure of the lattice may even change, but it must still remain rational (GBS, [7]).

Algorithm 2 Sprouting search algorithm framework.

```
init:     choose γ ∈ ℝ: γ ≥ 1;
          S := ∅, p := 0, x^o := x_0;
outer:    do begin
              if p = 0 then
                  choose H ∈ ℝ: H > 0;
              end
              choose a SDG G;
              choose h ∈ ℝ: h ≥ H;
              choose λ, Λ ∈ ℝ: (0 < λ ≤ Λ) ∧ (λ/Λ ≥ γ);
              let S be a subset of itself;
              quit := 0;
inner:        do begin
                  p := G(x^o, S);
descent:          if p ≠ 0 then
                      x^best := x^o;
                      if f(x^o + p) < f(x^o) − h then
                          x^best := x^o + p;
                          quit := 1;
                      end
```

```
                    S := S ∪ {x^o + p};
                end
                quit := 1;
            end
finite:     choose F ⊂ R^n: |F| < ∞;
            if |F| > 0 then
                x^e := arg min_{x∈F} f(x);
strict:         if f(x^e) < f(x^o) − h then
                    x^best := x^e;
                    quit := 1;
relaxed:        else if (quit = 1) ∧ (f(x^e) < f(x^best)) then
                    x^best := x^e;
                end
                S_f ⊆ {x : (x ∈ F) ∧ (f(x) ≥ f(x^o) − h)};
                S := S ∪ S_f;
            end
update:     x^o := x^best;
        while quit = 0;
    while stopping criteria not satisfied;
```

The algorithm consists of two loops. The inner loop searches for an origin $\boldsymbol{x}^{\mathrm{o}}$ from where it can't make any progress using a particular SDG G and a lower bound H on sufficient descent h. In related work h was the stepsize control parameter. Here however choosing the length of the step is in the domain of the SDG so h only determines the amount of descent required from a trial point to be accepted as the new origin. If G produces a trial step $\boldsymbol{p}$ such that $\boldsymbol{x}^{\mathrm{o}} + \boldsymbol{p}$ yields sufficient descent, the inner loop exits as soon as the results of a finite search process are evaluated. The finite search process is represented by evaluating the cost function at a set of points $\mathcal{F}$. The only requirement imposed on the finite search process is that the time from the start of the search process to its conclusion when the cost function is evaluated for all points from $\mathcal{F}$ (step 'finite') is finished in a finite amount of time. Assuming that every cost function evaluation takes a finite amount of time, this is also implied by the requirement $|\mathcal{F}| < \infty$.

There are two criteria for accepting the best trial point found by the finite search process. The first one requires sufficient descent and is applied after every failed trial step $\boldsymbol{p}$ produced by the SDG. The second one is milder and requires only simple descent. It is applied after every successful trial step p or when the SDG returns $\boldsymbol{p} = \boldsymbol{0}$ meaning that a positive basis around origin $\boldsymbol{x}^{\mathrm{o}}$ has been examined. If a point from $\mathcal{F}$ is accepted, the inner loop exits. If the point is accepted by the sufficient descent criterion, exiting the loop is forced by setting quit to one. In case the point is accepted by the milder

(second) criterion, the inner loop was about to exit anyway. The first criterion accepts a trial point $\boldsymbol{x}^{\mathrm{e}}$ examined by the finite search process if the sufficient descent condition is fulfilled with regard to the origin $\boldsymbol{x}^{\mathrm{o}}$. In practise this means that it may accept points with a higher cost function value than the one at $\boldsymbol{x}^{\mathrm{best}}$ as long as the descent is sufficient with regard to the origin $\boldsymbol{x}^{\mathrm{o}}$. To avoid increases of the cost function value one can use a more strict condition $(f(\boldsymbol{x}^{\mathrm{e}}) \leq f(\boldsymbol{x}^{\mathrm{o}}) - h) \wedge (f(\boldsymbol{x}^{\mathrm{e}}) \leq f(\boldsymbol{x}^{\mathrm{best}}))$.

The inner loop accumulates the examined trial steps in the set $\mathcal{S}$. This set is used by the SDG to calculate the next trial step. Some trial points examined by the finite search process also fail to satisfy the sufficient descent condition. The set $\mathcal{S}_{\mathrm{f}}$ represents the set of all trial steps from the current origin $\boldsymbol{x}^{\mathrm{o}}$ that could be considered as failed. A subset of these failed trial steps steps is also added to $\mathcal{S}$.

The outer loop selects a SDG and constants λ, Λ, h, and H. λ and Λ set the bounds on the region of space from where the trial points will be chosen in the inner loop. h determines the amount of descent that is considered sufficient. H enforces a lower bound on the amount of descent, and as it will be shown later, is a key part of the convergence proof. After these initial steps are done the inner loop is executed. The outer loop iterates until some point fulfilling the stopping condition is found.

Note that at this point the description of the strategy for choosing λ, Λ, h, and H is omitted. The requirements that this strategy must satisfy will be explained in the following section which deals with the convergence of the algorithm. Also omitted is the strategy for purging members from $\mathcal{S}$. Such a strategy is necessary in order to make the computations tractable.

4 Convergence

All convergence results are derived for the case when the outer loop (label 'outer') is infinite. Throughout the proof it is assumed that the there exists a compact set $\mathcal{C}$ so that $\mathcal{L}_f(\boldsymbol{x}_0) \subseteq \mathcal{C}$. Therefore $f(\boldsymbol{x})$ is bounded on $\mathcal{L}_f(\boldsymbol{x}_0)$. Since the algorithm accepts only points which decrease the cost function value with respect to the current origin $\boldsymbol{x}^{\mathrm{o}}$, all accepted points lie in this compact set.

LEMMA 4.1 *Suppose that h is bound from below by H and all SDGs used in the search process are feasible. Then the inner do-while loop's body executes a finite number of times before $\boldsymbol{p} = \boldsymbol{0}$ is generated.*

Proof Suppose this is not true. So the inner loop's body executes an infinite number of times without SDG returning $\boldsymbol{p} = \boldsymbol{0}$. If the SDG is feasible (definition 2.11), it produces $\boldsymbol{p} = \boldsymbol{0}$ in a finite number of iterations. So the inner loop

must exit before the SDG returns $\boldsymbol{p} = \boldsymbol{0}$. This can happen only if sufficient descent is obtained, either by the trial step or by the finite search process. So the function value decreases for at least $h \geq H$ every time the algorithm leaves the inner loop. Such decreases happen infinite many times, so $f(\boldsymbol{x})$ must decrease without bound, contradicting the assumption that $f(\boldsymbol{x})$ is continuously differentiable on a compact set. □

All changes to λ and Λ happen in the beginning of the outer loop's body. It directly follows from lemma 4.1 that the algorithm generates $\boldsymbol{p} = \boldsymbol{0}$ an infinite number of times. Let $\{\boldsymbol{x}_k^{\mathrm{o}}\}_{k=1}^{\infty}$ denote the sequence of search origins for which $\boldsymbol{p} = \boldsymbol{0}$ was generated, and $\{\lambda_k\}_{k=1}^{\infty}$, $\{\Lambda_k\}_{k=1}^{\infty}$, and $\{h_k\}_{k=1}^{\infty}$ the corresponding sequence of λ, Λ, and h values.

Lemma 4.2 *Suppose the algorithm generates $\boldsymbol{p} = \boldsymbol{0}$ infinitely many times and all SDGs used in the search process are feasible. Then provided that $\lim_{k\to\infty} \Lambda_k = 0$ and $\lim_{k\to\infty} h_k/\Lambda_k = 0$ hold, $\lim_{k\to\infty} \|\nabla f(\boldsymbol{x}_k^o)\| = 0$.*

Proof Let the $\{\mathcal{B}_k\}_{k=1}^{\infty}$ denote the sequence of positive bases found by the SDG that corresponds to the occurrences of $\boldsymbol{p} = \boldsymbol{0}$. According to lemma 2.12 the corresponding sequence $\{\mathcal{D}_k\}_{k=1}^{\infty}$ and all of its subsequences have at least one limit point and all limit points positively span $\mathbb{R}^n$. Since $\{\boldsymbol{x}_k^{\mathrm{o}}\}_{k=1}^{\infty}$ lies in a compact set, any of its subsequences also has at least one limit point. Choose a subsequence $\{\boldsymbol{x}_{j_k}^{\mathrm{o}}\}_{k=1}^{\infty}$ that converges to $\boldsymbol{x}_{\infty}^{o}$. Now from this subsequence choose another subsequence $\{\boldsymbol{x}_{i_k}^{\mathrm{o}}\}_{k=1}^{\infty}$ for which the corresponding subsequence $\{\mathcal{D}_{i_k}\}_{k=1}^{\infty}$ converges to $\mathcal{D}_{\infty}$. Since all trial steps from $\mathcal{B}_{i_k}$ failed to fulfill the sufficient descent condition with respect to the search origin $\boldsymbol{x}_{i_k}^{\mathrm{o}}$ the following inequality applies

$$f(\boldsymbol{x}_{i_k}^{\mathrm{o}} + \boldsymbol{b}) \geq f(\boldsymbol{x}_{i_k}^{\mathrm{o}}) - h_{i_k} \quad \forall \boldsymbol{b} \in \mathcal{B}_{i_k} \tag{9}$$

Let $\boldsymbol{q} = \boldsymbol{b}/\Lambda_{i_k}$ be some member of $\mathcal{D}_{i_k}$.

$$\begin{aligned} f(\boldsymbol{x}_{i_k}^{\mathrm{o}} + \Lambda_{i_k}\boldsymbol{q}) = \\ f(\boldsymbol{x}_{i_k}^{\mathrm{o}}) + \int_{l=0}^{\Lambda_{i_k}} \boldsymbol{q}^{\mathrm{T}}(\nabla f(\boldsymbol{x}_{i_k}^{\mathrm{o}} + l\boldsymbol{q}) - \nabla f(\boldsymbol{x}_{i_k}^{\mathrm{o}}) + \nabla f(\boldsymbol{x}_{i_k}^{\mathrm{o}}))dl = \\ f(\boldsymbol{x}_{i_k}^{\mathrm{o}}) + \Lambda_{i_k}\boldsymbol{q}^T\nabla f(\boldsymbol{x}_{i_k}^{\mathrm{o}}) + E, \quad \forall \boldsymbol{q} \in \mathcal{D}_{i_k}. \end{aligned} \tag{10}$$

Where

$$E = \int_{l=0}^{\Lambda_{i_k}} \boldsymbol{q}^T(\nabla f(\boldsymbol{x}_{i_k}^{\mathrm{o}} + l\boldsymbol{q}) - \nabla f(\boldsymbol{x}_{i_k}^{\mathrm{o}}))dl.$$

Now remember that $\|\boldsymbol{q}\|$ is bound from above ($\|\boldsymbol{q}\| = \|\boldsymbol{b}/\Lambda_{i_k}\| \leq 1$). From

the assumption of continuous differentiability of $f(\boldsymbol{x})$ over the initial level set it follows that $f(\boldsymbol{x})$ is uniformly continuous. In other words this means that for any $0 \leq l \leq \Lambda_{i_k}$ we have

$$\|\nabla f(\boldsymbol{x}^{\mathrm{o}}_{i_k} + l\boldsymbol{q}) - \nabla f(\boldsymbol{x}^{\mathrm{o}}_{i_k})\| \leq M.$$

When Λ approaches 0, M also goes to 0.

$$\lim_{\Lambda_{i_k} \to 0} M = 0.$$

So $|E|$ is bound from above

$$|E| \leq \int_{l=0}^{\Lambda_{i_k}} M dl = M\Lambda_{i_k}. \tag{11}$$

By joining (9), (10), and (11) the following relation is obtained

$$\boldsymbol{q}^{\mathrm{T}} \nabla f(\boldsymbol{x}^{\mathrm{o}}_{i_k}) + M \geq -h_{i_k}/\Lambda_{i_k}, \quad \forall \boldsymbol{q} \in \mathcal{D}_{i_k}.$$

When i_k approaches ∞ the previous inequality simplifies to

$$\boldsymbol{q}^{\mathrm{T}}_{\infty} \nabla f(\boldsymbol{x}^{\mathrm{o}}_{\infty}) \geq 0, \quad \forall \boldsymbol{q}_{\infty} \in \mathcal{D}_{\infty}.$$

$\mathcal{D}_{\infty}$ positively spans $\mathbb{R}^n$ (lemma 2.12). Together with lemma 2.3 this results in

$$\|\nabla f(\boldsymbol{x}^{\mathrm{o}}_{\infty})\| = 0.$$

$\boldsymbol{x}^{\mathrm{o}}_{\infty}$ was an arbitrary limit point so the last result holds for any limit point of $\{\boldsymbol{x}^{\mathrm{o}}_k\}_{k=1}^{\infty}$. Consequently

$$\lim_{k \to \infty} \|\nabla f(\boldsymbol{x}^{\mathrm{o}}_k)\| = 0.$$

□

THEOREM 4.3 *Suppose that the value of H changes only when $\boldsymbol{p} = \boldsymbol{0}$ is generated by the SDG, $lim_{k\to\infty}\Lambda_k = 0$, and $lim_{k\to\infty}h_k/\Lambda_k = 0$. Then* $\lim_{k\to\infty} \|\nabla f(\boldsymbol{x}^{\mathrm{o}}_k)\| = 0$.

Proof The assumption on the behaviour of H together with lemma 4.1 guarantees that the SDG generates $\boldsymbol{p} = \boldsymbol{0}$ an infinite number of times. If we also take into account lemma 4.2, we obtain $\lim_{k\to\infty} \|\nabla f(\boldsymbol{x}^{\mathrm{o}}_k)\| = 0$. □

The last result extends the results of Coope and Price, in the sense that the step length control is no longer directly coupled to the sufficient descent control. Of course the step length (Λ_k) must still approach zero, since the algorithm has to probe a sufficiently small neighborhood of $\boldsymbol{x}^{\mathrm{o}}$ in order to establish that $\|\nabla f(\boldsymbol{x}^{\mathrm{o}})\| = 0$. The amount of descent (h_k) which is considered sufficient, must also approach zero, otherwise the cost function value would have to descend without any lower bound. Both requirements are quite intuitive. Not so obvious is the requirement that the amount of sufficient descent (h) must approach zero faster than the stepsize (Λ). This requirement is fulfilled by all direct search methods whose convergence is based on sufficient descent. It can be found in the formulation of the frame-based methods [8], where it appears in the form $h = \alpha\Lambda^{1+\beta}$, $\alpha, \beta > 0$. Lucidi and Sciandrone [9] define a function which connects h to Λ ($h = o(\Lambda)$). The function must fulfill the requirement $o(x)/x \to 0$ as x approaches zero. Garcia-Palomares and Rodriguez [10] define a class of direct search methods with a sufficient descent requirement and also use a similar formulation.

The advantage of the above described framework becomes evident when one tries to use it for defining asynchronous parallel direct search optimization algorithms. The next section addresses the issues associated with it.

5 Parallel asynchronous algorithm

The previous section establishes the convergence of the algorithms conforming to the framework of algorithm *2* (sprouting search). In cases when one cost function evaluation takes a substantial amount of time methods for dividing the work among several processing units become attractive. As the underlying hardware a cluster of ordinary computers (i.e. PCs) connected by a LAN can be used. This was the case with APPS. There are several things we expect from a parallel algorithm:

- distribution of work,
- fault tolerance, and
- run-time scalability.

Since for a particular processing unit all other processing units represent a finite search process, the convergence theory developed in the previous section can also be applied to the asynchronous parallel case. The limit points of the search origin for every processing unit, executing an algorithm conforming to the framework, are also stationary points of the cost function.

5.1 *Distribution of work*

The simplest way for implementing an asynchronous parallel algorithm is to use a toolkit like PVM [30]. PVM takes care of collecting messages from and sending messages to processes on different computers. When the program needs to respond to all past messages it simply reads them out from its input queue and processes them. PVM takes care that all incoming messages go to the input queue of the process and that no message gets lost. If there is no need for stopping an ongoing CF evaluation as a response to an incoming message (like in APPS) there is no need for a worker to have multiple parallel paths of execution (multiple OS threads or processes). On the other hand when a worker finishes a CF evaluation the obtained result is immediately broadcasted to other workers in form of a message. Every received point can be treated as a result obtained by some finite search process.

The finite search process and the processing of received messages takes place after every trial step evaluation. The requirements that have to be fulfilled by the results obtained from the finite search process in order to be accepted, depend on the state the search is currently in. They are milder (simple descent) when the inner loop is about to terminate. This resembles the FBS except that FBS has two finite search processes, one in the inner and one in the outer loop of the algorithm.

Nothing was said about the finite search process except that it must terminate in a finite amount of time. It would of course be of advantage if the finite search process was capable of using the information from received messages to help guide its own decisions. Furthermore, the finite search process could also produce and send out messages to notify other processing units of its progress even before it is finished. In the presented convergence analysis no limitation is imposed on the way the finite search uses the received result messages or produces new outgoing result messages, except that the number of produced messages must remain finite.

5.2 *Fault tolerance and scalability*

Since all processing units are capable of finding a minimum on their own, failures of processing units don't prevent the cluster from progressing toward a local minimum of the cost function. As long as there is at least one processing unit, the progress of the search is guaranteed. Of course it can be expected that the time needed to find a minimum increases when the number of processing units participating in the search decreases. The presented framework has built-in fault tolerance. Any algorithm conforming to the framework will exhibit fault tolerance in the sense that failures of individual processing units can't stop the system from progressing toward a solution.

A further advantage of the presented framework is its capability to change the number of processing units while the search is in progress. All processing units keep a local list of cluster members (member addresses). There are two operations that are performed on this list.

- When a failure of an individual processing unit is detected (i.e. when an exception is detected) the respective processing unit's address is deleted from the local list of cluster members.
- When a notification of the presence of a new processing unit is received, the respective processing unit is added to the local list.

5.3 *Heterogeneous distributed algorithms and hybrid algorithms*

The loose requirements imposed on the finite search process do not prohibit the use of heterogeneous distributed algorithms. By heterogeneous distributed algorithms we mean that different processing units execute different search algorithms. As long as we have at least one cluster member executing an algorithm conforming to the framework presented in this paper, the sequence of points examined in the inner loop of the algorithm will have at least one cluster point and every cluster point will be a stationary point of the cost function $f(\boldsymbol{x})$. One can for instance have a couple of workers executing some global search algorithm (e.g. simulated annealing or genetic algorithm) which occasionally incorporates points received from other workers into its population. The rest of the cluster executes an algorithm conforming to the presented framework and refines the results obtained by the global search.

On the other hand an algorithm conforming to the framework of algorithm *2* doesn't have to obtain the points in set $\mathcal{F}$ solely from other processing units. When it comes to the evaluation of the set, it can add points to it by running any finite process. This enables us to incorporate existing optimization algorithms into the framework of algorithm *2* thus producing hybrid algorithms like [13, 14].

5.4 *Comparison with other asynchronous direct search algorithms*

To our best knowledge only APPS qualifies in this category. APPS achieves fault tolerance by distributing the search directions among workers. Every worker handles one direction of the search. The cluster of computers is capable of finding a local minimum as long as the set of directions searched by the workers positively spans $\mathbb{R}^n$. APPS constantly checks whether the failed steps form a positive basis. Adapting the search directions to the cost function is not simple since it requires a large amount of coordination between individual workers. Scalability is not mentioned in any of the papers describing APPS.

In the presented sprouting search framework the notion of a SDG is introduced, which stands for both choosing consecutive directions that positively span $\mathbb{R}^n$, and adapting the search step size to the local function behaviour. Furthermore due to mild restrictions imposed on the SDG, it is possible to incorporate approximate gradient information in the search. The gradient can be approximated by using for instance the simplex gradient approach (see [31] or [32]). The obtained gradient approximation can then be used to allign the set of directions generated by the SDG, it may be used to conduct a line search in the finite search process, or it may also be used to update the variable metric. None of these possibilities is offered by the rigid framework of APPS.

In APPS there exists a strong connection between the number of workers and the number of search directions. A particular worker is responsible for searching in its designated direction. The sprouting search framework has no such limitation. Any worker can look in any direction. The number of workers can be smaller than the number of directions and it can even change during the search without affecting the convergence properties of the algorithm. The SDG of every worker is responsible for choosing the search direction in such a way that it eventually causes the algorithm to converge. As it was shown in the previous sections this can easily be achieved by placing some simple restrictions on the SDGs.

Sprouting search offers a flexible framework for asynchronous parallel optimization. It is completely peer-to-peer and has no centralized control. As long as there is at least one worker in the cluster, the search continues. Even if the cluster was divided in several disjoint clusters, every part would continue the search on its own and would eventually reach a stationary point of the cost function.

6 Example: hybrid simplex-sprouting algorithm.

The presented framework was tested using MATLAB (TM) [33] with the DP [34] toolbox. The DP toolbox provides access to most of the PVM [30] (Parallel Virtual Machine) functionality from within MATLAB. The Nelder-Mead algorithm [1] was used as the basis of the new hybrid algorithm that was run on a cluster of 6 AMD ATHLON 2100XP computers.

Algorithm 3 A single step of the modified Nelder-Mead simplex algorithm (without the shrink step).

Order simplex vertices so that $f_{\mathrm{s}}^1 \leq f_{\mathrm{s}}^2 \leq ... \leq f_{\mathrm{s}}^{n+1}$;
for $i = 1, 2, ..., n_{\mathrm{mirr}}$ **do**
$\boldsymbol{x}_{\mathrm{s}}^{\mathrm{c}} := 1/(n+1-i) \sum_{j=1}^{n+1-i} \boldsymbol{x}_{\mathrm{s}}^j$;

$\boldsymbol{x}_\mathrm{s}^\mathrm{r} := \boldsymbol{x}_\mathrm{s}^\mathrm{c} - \gamma_\mathrm{r}(\boldsymbol{x}_\mathrm{s}^{n+2-i} - \boldsymbol{x}_\mathrm{s}^\mathrm{c})$;
if $f(\boldsymbol{x}_\mathrm{s}^\mathrm{r}) < f_\mathrm{s}^1$ **then**
$\boldsymbol{x}_\mathrm{s}^\mathrm{e} := \boldsymbol{x}_\mathrm{s}^\mathrm{c} - \gamma_\mathrm{e}(\boldsymbol{x}_\mathrm{s}^{n+2-i} - \boldsymbol{x}_\mathrm{s}^\mathrm{c})$;
if $f(\boldsymbol{x}_\mathrm{s}^\mathrm{e}) < f(\boldsymbol{x}_\mathrm{s}^\mathrm{r})$ **then**
$\boldsymbol{x}_\mathrm{s}^{n+2-i} := \boldsymbol{x}_\mathrm{s}^\mathrm{e}$;
else
$\boldsymbol{x}_\mathrm{s}^{n+2-i} := \boldsymbol{x}_\mathrm{s}^\mathrm{r}$;
end
else if $f_\mathrm{s}^1 \leq f_\mathrm{s}^\mathrm{r} < f_\mathrm{s}^{n+1-i}$ **then**
$\boldsymbol{x}_\mathrm{s}^{n+2-i} := \boldsymbol{x}_\mathrm{s}^\mathrm{r}$;
else if $f_\mathrm{s}^{n+1-i} \leq f_\mathrm{s}^\mathrm{r} < f_\mathrm{s}^{n+2-i}$ **then**
$x_\mathrm{s}^\mathrm{oc} := \boldsymbol{x}_\mathrm{s}^\mathrm{c} - \gamma_\mathrm{oc}(\boldsymbol{x}_\mathrm{s}^{n+2-i} - \boldsymbol{x}_\mathrm{s}^\mathrm{c})$;
if $f(\boldsymbol{x}_\mathrm{s}^\mathrm{oc}) \leq f_\mathrm{s}^{n+2-i}$ **then**
$\boldsymbol{x}_\mathrm{s}^{n+2-i} := \boldsymbol{x}_\mathrm{s}^\mathrm{oc}$;
end
else if $f_\mathrm{s}^{n+2-i} \leq f_\mathrm{s}^\mathrm{r}$ **then**
$\boldsymbol{x}_\mathrm{s}^\mathrm{ic} := \boldsymbol{x}_\mathrm{s}^\mathrm{c} - \gamma_\mathrm{ic}(\boldsymbol{x}_\mathrm{s}^{n+2-i} - \boldsymbol{x}_\mathrm{s}^\mathrm{c})$;
if $f(\boldsymbol{x}_\mathrm{s}^\mathrm{ic}) \leq f_\mathrm{s}^{n+2-i}$ **then**
$\boldsymbol{x}_\mathrm{s}^{n+2-i} := \boldsymbol{x}_\mathrm{s}^\mathrm{ic}$;
end
end
end

The original Nelder-Mead algorithm was modified in the sense that it moves multiple points of simplex $\{\boldsymbol{x}_\mathrm{s}^1, \boldsymbol{x}_\mathrm{s}^2, ..., \boldsymbol{x}_\mathrm{s}^{n+1}\}$ in a single step (see algorithm *3*). If $n_\mathrm{mirr} = 1$ the algorithm becomes the original Nelder-Mead algorithm. Note that there are no shrink steps in this modified algorithm. The value of the reflection (γ_r), expansion (γ_e), inner contraction (γ_ic), outer contraction (γ_oc), and shrink (γ_s) coefficient is 1, 2, 0.5, −0.5, and 0.5, respectively.

For incorporating points into the simplex algorithm *4* was used. Every received point is represented by a 3-tuple consisting of the point coordinates, the cost function value, and point index in the sender's simplex.

Algorithm 4 Incorporating a point into the simplex.

Let every received point be represented by a 3-tuple $(\boldsymbol{x}, f(\boldsymbol{x}), j)$.
Order simplex vertices so that $f_\mathrm{s}^1 \leq f_\mathrm{s}^2 \leq ... \leq f_\mathrm{s}^{n+1}$;
$\Delta := (f_\mathrm{s}^{n+1} - f_\mathrm{s}^1)/n$;
for $i = n+1, n, ..., 2$ **do**
Of all the received points with $j = i$,
let $\boldsymbol{x}$ denote the one with the lowest f;
if such $\boldsymbol{x}$ exists $\wedge$ $f(\boldsymbol{x}) < f_\mathrm{s}^1 - \Delta$ **then**

$\boldsymbol{x}_\mathrm{s}^i := \boldsymbol{x}$;
end
end

Computers in the cluster execute the hybrid algorithm *5*. At several points in the algorithm simplex is ordered in such a manner that the cost function values $f_\mathrm{s}^i = f(\boldsymbol{x}_\mathrm{s}^i)$ form a nondecreasing sequence. In the beginning an initial simplex is constructed, the cost function at its vertices is evaluated, and the corresponding 3-tuples are broadcasted. Note that the broadcasted point index is zero, meaning that these points (when received by other computers) can be used only in the sprouting part of the algorithm. For determining the initial Δ and h the simplex is first ordered. upon which h is calculated as $(f_\mathrm{s}^{n+1} - f_\mathrm{s}^1)/(100n)$. For Δ the simplex side vectors $\boldsymbol{v}_\mathrm{s}^i = \boldsymbol{x}_\mathrm{s}^{i+1} - \boldsymbol{x}_\mathrm{s}^1$, $i = 1, 2, ..., n$ are first evaluated upon which Δ is set to $1/10 \min_{i=1,2,...,n} \|\boldsymbol{v}_\mathrm{s}^i\|$.

Algorithm 5 Hybrid simplex-sprouting algorithm.

Choose initial simplex vertices $\{\boldsymbol{x}_\mathrm{s}^1, \boldsymbol{x}_\mathrm{s}^2, ..., \boldsymbol{x}_\mathrm{s}^{n+1}\}$.
Evaluate f at simplex vertices to obtain $\{f_\mathrm{s}^1, f_\mathrm{s}^2, ..., f_\mathrm{s}^{n+1}\}$.
Broadcast 3-tuples $(\boldsymbol{x}_\mathrm{s}^i, f_\mathrm{s}^i, 0)$ for $i = 1, 2, ..., n+1$.
Choose initial Δ and h, clear the set of received 3-tuples.
while $f > f_\mathrm{stop}$ **do**
  do
    $f_\mathrm{s}^{\max 1} := \max_i f_\mathrm{s}^i$;
    Perform one simplex step (algorithm *3*).
    $f_\mathrm{s}^{\max 2} := \max_i f_\mathrm{s}^i$;
    Broadcast evaluated points as 3-tuples.
    if $f_\mathrm{s}^{\max 2} - f_\mathrm{s}^{\max 1} \leq -h$ **then**
      Incorporate received points into the simplex (algorithm *4*).
      Clear the set of received 3-tuples.
    end
  while $f_\mathrm{s}^{\max 2} - f_\mathrm{s}^{\max 1} \leq -h$;
  Order simplex vertices so that $f_\mathrm{s}^1 \leq f_\mathrm{s}^2 \leq ... \leq f_\mathrm{s}^{n+1}$;
  $\boldsymbol{x}^\mathrm{o} := \boldsymbol{x}_\mathrm{s}^1$;
  Calculate linear basis $\mathcal{B}_\mathrm{L}$ based on current simplex shape.
  $\mathcal{S} := \{\boldsymbol{x}_\mathrm{s}^2, \boldsymbol{x}_\mathrm{s}^3, ..., \boldsymbol{x}_\mathrm{s}^{n+1}\}$; $\mathcal{B} := \mathcal{B}_\mathrm{L} \cup -\mathcal{B}_\mathrm{L}$;
  exit := 0;
  while exit $\neq 1$ **do**
    Add points from the set of received 3-tuples to set $\mathcal{S}$.
    Clear the set of received 3-tuples.
    $\boldsymbol{x}^\mathrm{min} := \min_{\boldsymbol{x} \in \mathcal{S}} f(\boldsymbol{x})$;
    Clean up set $\mathcal{S}$.
    if $f(\boldsymbol{x}^\mathrm{min}) - f(\boldsymbol{x}^\mathrm{o}) \leq -h$ **then**

```
            x_s^1 := x^min;
            exit := 1;
        else
            p = G(x^o, S).
            if p ≠ 0 then
                if f(x^o + p) − f(x^o) ≤ −h then
                    x_s^1 := x^min;
                    exit := 1;
                else
                    S := S ∪ {x^o + p};
                end
                Broadcast 3-tuple (x^o + p, f(x^o + p), 0).
            else
                B_L := γ_s B_L; Δ := γ_s Δ; h := (γ_s)^ν h;
                Replace simplex with {x^o} ∪ x^o + B_L or {x^o} ∪ x^o − B_L.
                Broadcast 3-tuples (x_s^i, f_s^i, 0) for i = 1, 2, ..., n + 1.
                exit := 1;
            end
        end
    end
end
```

In the main loop every pass of algorithm *3* is followed by the broadcasting of the examined points to other workers. The broadcasted point index is the index of the point that was moved by the modified simplex algorithm ($n+2-i$ in algorithm *3*). If algorithm *3* decreases the highest cost function value in the simplex by at least h (sufficient descent) the points from the received 3-tuples are incorporated into the simplex (algorithm *4*). If not, the sprouting part of the algorithm is entered.

In the sprouting part of the algorithm the simplex is first ordered and its n side vectors ($\boldsymbol{v}_{\mathrm{s}}^i$) are evaluated. The side vectors are ordered so that $\boldsymbol{v}_{\mathrm{s}}^1$ is the longest and $\boldsymbol{v}_{\mathrm{s}}^n$ the shortest side vector. A matrix $\mathbf{V} = [\boldsymbol{v}_{\mathrm{s}}^1 \boldsymbol{v}_{\mathrm{s}}^2 ... \boldsymbol{v}_{\mathrm{s}}^n]$ with simplex side vectors for columns is formed and decomposed by means of QR decomposition ($\mathbf{V} = \mathbf{QR}$). Reshaped side vectors are obtained from diagonal elements $\mathbf{R}_{ii}$ of matrix $\mathbf{R}$ and columns $\boldsymbol{q}_i$ of matrix $\mathbf{Q}$ using the following formula: $\boldsymbol{q}_i \mathrm{sign}(\mathbf{R}_{ii}) \max(1.1\Delta, \min(|\mathbf{R}_{ii}|, 0.9 \cdot 10^{14}\Delta))$. If $\mathbf{R}_{ii}$ is zero, $\mathrm{sign}(\mathbf{R}_{ii})$ is considered to be 1. $\mathcal{B}_{\mathrm{L}}$ denotes the set of reshaped side vectors. A similar reshape procedure was used by Byatt [14] in his convergent variant of the Nelder-Mead simplex algorithm.

Positive basis $\mathcal{B} = \mathcal{B}_{\mathrm{L}} \cup -\mathcal{B}_{\mathrm{L}}$ is used by the SDG. The best point of the simplex ($\boldsymbol{x}_{\mathrm{s}}^1$) is the origin $\boldsymbol{x}^{\mathrm{o}}$. Before the SDG loop is entered, $\mathcal{S}$ is cleaned up. All points that produce sufficient descent with respect to the origin are removed.

The remaining points are transformed to linear basis $\mathcal{B}_\mathrm{L}$. The SDG operates in this basis. Points whose squared sum of coordinate distances from the origin in this new basis is below 0.25 or above 4 are also removed. Effectively this keeps only points positioned between two ellipsoids around the origin.

In the beginning of the SDG loop the received 3-tuples are checked if they contain a point that improves the cost function at the origin by more than h. If such a point exists the best point of the simplex ($\boldsymbol{x}_\mathrm{s}^1$) is replaced by it and the algorithm returns to the Nelder-Mead part. If not the SDG generates a search direction $\boldsymbol{p}$.

If $\boldsymbol{p}$ is a nonzero vector, the cost function is evaluated at $\boldsymbol{x}^\mathrm{o}+\boldsymbol{p}$. If sufficient descent is obtained (with respect to $\boldsymbol{x}^\mathrm{o}$), the point replaces the best point of the simplex ($\boldsymbol{x}_\mathrm{s}^1$) and the algorithm returns back to the Nelder-Mead part. If not, $\boldsymbol{x}^\mathrm{o}+\boldsymbol{p}$ is added to $\mathcal{S}$. In both cases the 3-tuple $(\boldsymbol{x}^\mathrm{o}+\boldsymbol{p}, f(\boldsymbol{x}^\mathrm{o}+\boldsymbol{p}), 0)$ is broadcasted to other computers.

If however the SDG returns a zero vector, the basis $\mathcal{B}_\mathrm{L}$ shrinks by γ_s. Δ and h are also updated ($\nu = 2$). Two candidate simplices are generated: $\{\boldsymbol{x}^\mathrm{o}\}\cup\boldsymbol{x}^\mathrm{o}+\mathcal{B}_\mathrm{L}$ and $\{\boldsymbol{x}^\mathrm{o}\}\cup\boldsymbol{x}^\mathrm{o}-\mathcal{B}_\mathrm{L}$. The simplex with the lowest cost function value (excluding the one at $\boldsymbol{x}^\mathrm{o}$) is chosen as the new simplex. 3-tuples representing the points of this simplex are broadcasted to other computers and the algorithm returns to the Nelder-Mead part.

It is worth noting that the SDG works with vectors expressed in linear basis $\mathcal{B}_\mathrm{L}$. The corresponding condition in algorithm *1* is $\|\boldsymbol{y}\| > C \ \ \vee \|\mathbf{V}\boldsymbol{y}-\boldsymbol{b}^i\| > D$ ($C = 10$, $D = 10^{-3}$). In our case $\lambda = 0.5$, $\Lambda = 2$, and $\|\boldsymbol{b}^\mathrm{max}\| = \|\boldsymbol{b}^\mathrm{min}\| = 1$ so the simplification makes sense. The second part of the condition ensures that the basis direction $\boldsymbol{b}^i$ is returned as $\boldsymbol{p}$ if it can't be expressed in the NLLS sense with vectors from set $\mathcal{S}-\boldsymbol{x}^\mathrm{o}$ (within sufficient accuracy prescribed by D).

For numerical testing the algorithm was run 10 times for every combination of a test function and a number of processing units. Functions from [35] and [36] were used. The initial simplex was chosen to comprise the initial point $\boldsymbol{x}_0$ and n additional random points chosen from a box around the initial point. The box size was $0.05\|\boldsymbol{x}_0\|$ (0.00025 if $\|\boldsymbol{x}_0\|$ was 0). The number of mirrored points was $n_\mathrm{mirr} = \lceil \min(n,p)/2 \rceil$ where p was the number of workers. The effect of varying CF evaluation time, that causes the synchronous algorithms to waste a lot of time at synchronisation points, was emphasized by adding a random delay after every CF evaluation.

In order to ensure that the results obtained with different numbers of computers are comparable, the initial simplices for individual computers were chosen in the following manner. For a particular test function and a particular run (out of the 10 runs tried) simplices $\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_p$ were used for computers $1, 2, ..., p$. If the run was performed with one computer, only $\mathcal{A}_1$ was used in the optimization. If the run was performed with two computers, simplices $\mathcal{A}_1$ and

Table 1. Timing results for an algorithm conforming to the framework of algorithm 3. Simulation was stopped after f_{stop} was reached. Values for $p = 1$ are in seconds, whereas other values represent speedups.

Test function	n	f_{stop}	$p = 1$	$p = 2$	$p = 4$	$p = 6$
trid	4	10^{-6}	12.1	1.45	1.85	1.93
band	4	10^{-6}	12.6	1.44	1.68	1.95
chebyquad	4	10^{-6}	18.8	1.34	1.47	1.60
sing	4	10^{-6}	28.0	1.50	2.60	3.13
trig	4	10^{-12}	17.2	1.26	1.70	1.78
vardim	4	10^{-6}	30.6	1.64	2.62	2.89
fminsurf	16	10^{-3}	117.1	1.41	2.38	3.05
gulf	3	10^{-6}	23.2	1.28	2.29	2.69
morebv	10	10^{-6}	66.9	1.54	2.50	3.20
osbornea	5	10^{-10}	88.1	1.27	1.82	2.01
osborneb	11	10^{-7}	350.3	1.57	3.40	4.52
yfitu	3	10^{-6}	31.5	1.37	1.47	1.76
meyer3	3	10^{-4}	158.6	1.23	1.27	1.52
helix	3	10^{-12}	25.7	1.51	2.09	2.15
quadratic	8	10^{-6}	33.4	1.31	2.02	2.27
quadratic	16	10^{-6}	97.6	1.30	1.91	2.32
almost	7	10^{-6}	117.6	2.32	4.40	5.87
almost	15	10^{-6}	1070.7	1.70	5.68	9.41
rosex	8	10^{-6}	270.3	1.50	3.23	4.52
rosex	16	10^{-6}	1160.6	1.23	2.61	3.59
vardim	8	10^{-6}	332.2	2.08	6.32	7.96
singx	8	10^{-6}	173.2	1.64	3.72	4.55
singx	16	10^{-6}	644.9	1.42	2.90	3.69
palmer1c	8	10^{-7}	551.8	1.20	3.87	5.08
palmer1d	7	10^{-6}	286.1	1.24	3.32	3.95
palmer2c	8	10^{-7}	523.1	1.10	3.92	5.29
palmer3c	8	10^{-7}	525.5	1.38	4.24	5.48

$\mathcal{A}_2$ were used (one for every computer), etc. Optimization was stopped when a particular cost function value (f_{stop}) was reached. The run was also stopped if the amount of sufficient descent became less than $10^{-14} \min_{i=1,2,\ldots,n+1} |f_{\text{s}}^i|$. The average run time for the runs that reached the target cost is listed in table 1.

From table 1 it can be seen that the speedup depends on the problem. In all cases a speedup was obtained that increased with the growing number of workers. There were only a few failed runs. Failures were observed with the osborneb test function (one for $p = 1$ and $p = 2$, and two for $p = 4$) and with the palmer1c function (one for $p = 1$). No failures were observed with $p = 6$ workers.

In order to demonstrate the scalability and fault tolerance of the parallel algorithm the following experiment was performed. After the cost function comes halfway from the value at the starting point toward f_{stop} the number of active workers is suddenly decreased from six to three (or increased from three to six). Table 2 lists the results of the experiment averaged across 10 runs for four different test functions.

Table 2. Timing results (in seconds) for adding/removing processing units from the cluster while optimization is in progress.

Test function	n	f_{stop}	$p = 6$	$p : 6 \to 3$	$p = 3$	$p : 3 \to 6$
quadratic	8	10^{-16}	39.8	43.9	49.2	48.7
rosex	8	10^{-16}	114.1	144.8	191.2	172.4
vardim	8	10^{-16}	84.6	96.8	114.0	110.6
singx	8	10^{-16}	101.8	160.7	189.5	136.9

The values in the fourth column of table 2 represent the timing results for 6 workers working in parallel. The fifth column represents the timing results for the case when half of the workers leave the cluster as the cost function comes halfway toward its final value. As it can be seen from the table the final value is reached, but it takes more time to reach it.

The seventh column of table 2 represents the case when the number of workers increases from three to six as the cost function comes halfway toward its final value. In this case the timing results from the seventh column are better than those from the sixth column (that represent the average timing for $p = 3$). The improvement is however not as big as is the drawback when the number of workers decreases from six to three. This can be explained by the fact that a worker leaving the cluster has immediate effect on the speed of progress, but a worker that joins the cluster must first adapt to the situation the search is currently in, and thus its effect is not immediate.

7 Conclusion

The sprouting search framework for (asynchronous parallel) direct search was presented. The notion of a search direction generator, generalizing the process of choosing a trial step, was introduced. A proof of framework's convergence under mild assumptions for continuously differentiable cost functions with compact level sets was presented. The finiteness of the inner loop is guaranteed by the sufficient descent condition and the finite number of iterations needed for a feasible search direction generator to produce a positive spanning set. In the framework the trial step size control is decoupled from the sufficient descent control. The only requirement is that the amount of sufficient descent approaches zero faster than the trial step size.

The framework is very flexible and within its boundaries one can define a wide variety of different asynchronous parallel optimization algorithms. Such algorithms are especially well suited for clusters of workstations. Fault tolerance is built into the framework in the sense that failures of individual processing units can't stop the algorithm from progressing toward a solution. Another benefit is the framework's capability to change the number of process-

ing units while the search is in progress. The limit points for every processing unit executing an algorithm conforming to the aforementioned framework are also stationary points of the cost function.

The framework's generality enables us to build heterogeneous clusters of workers where workers execute different algorithms and exchange points. As long as there is at least one worker running a sprouting search algorithm, that worker will exhibit convergence and by point exchange force other workers to converge too. Furthermore it is possible to construct hybrid algorithms consisting of sprouting search and some well established algorithm that may even lack convergence theory. The theoretical proofs for the sprouting search algorithm also holds for the hybrid algorithm.

The above mentioned properties were demonstrated by constructing a hybrid algorithm with Nelder-Mead simplex algorithm for its basis. Speedup was demonstrated by running the algorithm on several test problems and with different numbers of workers in the cluster. Fault tolerance and dynamic cluster sizing were demonstrated by removing and adding workers to the cluster while the search was in progress. In both cases the algorithm was capable of reaching the prescribed accuracy in spite of failures and changes in the cluster. The timing results also confirmed the expected effect of adding (removing) workers to (from) the cluster.

Acknowledgments

The authors wish to thank M. Fukushima, M. Sciandrone, D. Byatt, and C. Price for the electronic versions of their technical reports and papers.

The research was co-funded by the Ministry of Education, Science, and Sport (Ministrstvo za Šolstvo, Znanost in Šport) of the Republic of Slovenia through the programme P2-0246 Algorithms and optimization methods in telecommunications.

References

[1] Nelder, J. A. and Mead, R., 1965, A simplex method for function minimization. *The computer Journal*, **7**, 308–313.

[2] McKinnon, K. I. M., 1998, Convergence of the Nelder-Mead simplex method to a nonstationary point. *SIAM J. Optim.*, **9**, 148–158.

[3] Wright, M. H., 1996, Direct search methods: once scorned, now respectable. In: Griffiths, D. F. and Watson, G. A. (Eds.), *Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis*, 191–208 (Boston: Addison Wesley Longman).

[4] Torczon, V. J., 1989, Multi-Directional Search: A Direct Search Algorithm for Parallel Machines. PhD thesis, Department of Mathematical Sciences, Rice University, TX, USA.

[5] Torczon, V. J., 1991, On the convergence of the multidirectional search algorithm. *SIAM J. Optim.*, **1**, 123–145.

[6] Torczon, V. J., 1997, On the convergence of pattern search methods. *SIAM J. Optim.*, **7**, 1–25.

[7] Coope, I. D. and Price, C. J., 2001, On the convergence of grid-based methods for unconstrained optimization. *SIAM J. Optim.*, **11**, 859–869.
[8] Coope, I. D. and Price, C. J., 2000, Frame based methods for unconstrained optimization. *J. Optim. Theory Appl.*, **107**, 261–274.
[9] Lucidi, S., and Sciandrone, M., 2002, On the global convergence of derivative-free methods for unconstrained optimization. *SIAM J. Optim.*, **13**, 97–116.
[10] Garcia-Palomares, U. M. and Rodriguez, J. F., 2002, New sequential and parallel derivative-free algorithms for unconstrained minimization. *SIAM J. Optim.*, **13**, 79–96.
[11] Hooke, R. and Jeeves, T. A., 1961, 'Direct search' solution of numerical and statistical problems. *Journal of the Association for Computing Machinery*, **8**, 212–229.
[12] Coope, I. D. and Price, C. J., 2000, A direct search conjugate directions algorithm for unconstrained minimization. *ANZIAM J.*, **42**, C478–C498.
[13] Byatt, D., 2000, A convergent variant of the Nelder-Mead algorithm. Master's thesis, Mathematics and Statistics Department, University of Canterbury, NZ.
[14] Price, C. J., Coope, I. D., and Byatt, D., 2002, A convergent variant of the Nelder-Mead algorithm. *J. Optim. Theory Appl.*, **113**, 5–19.
[15] Powell, M. J. D., 1998, Direct search algorithms for optimization calculations. *Acta Numer.*, **7**, 287–336.
[16] Lewis, R. M., Torczon, V. J., and Trosset, M. W., 2000, Direct search methods: Then and now. *J. Comput. Appl. Math.*, **124**, 191–207.
[17] Dennis, J. E. and Torczon, V. J., 1991, Direct search methods on parallel machines. *SIAM J. Optim.*, **1**, 448–474.
[18] Mangasarian, O. P., 1995, Parallel gradient distribution in unconstrained optimization. *SIAM J. Control Optim.*, **33**, 1916–1925.
[19] Ferris, M. C. and Mangasarian, O. P., 1994, Parallel variable distribution. *SIAM J. Optim.*, **4**, 102–126.
[20] Solodov, M. V., 1997, New inexact parallel variable distribution algorithms. *Comput. Optim. Appl.*, **7**, 165–182.
[21] Fukushima, M., 1998, Parallel variable transformation for unconstrained optimization. *SIAM J. Optim.*, **8**, 658–672.
[22] Dennis, J. E. and Torczon, V. J., 1988, Parallel implementations of the nelder-mead simplex algorithm for unconstrained optimization. *Proceedings of the SPIE - The International Society for Optical Engineering*, **880**, 187–191.
[23] Coetzee, L. and Botha, E. C., 1998, The parallel downhill simplex algorithm for unconstrained optimisation. *Concurrency: Practice and Experience*, **10**, 121–137.
[24] Hough, P. D., Kolda, T. G., and Torczon, V. J., 2001, Asynchronous parallel pattern search for nonlinear optimization. *SIAM J. Sci. Comput.*, **23**, 134–156.
[25] Kolda, T. G. and Torczon, V. J., 2004, On the convergence of asynchronous parallel pattern search. *SIAM J. Optim.*, **14**, 939–964.
[26] Kolda, T. G. and Torczon, V. J., 2002, Understanding asynchronous parallel pattern search. Technical Report SAND2001-8695, Sandia National Laboratories, Livermore, CA, USA.
[27] Bertsekas, D. and Tsiksiklis, J., 2001, *Parallel and Distributed Computation: Numerical Methods* (Englewood Cliffs: Prentice-Hall).
[28] Davis, C., 1954, Theory of positive linear dependence. *Amer. J. Math.*, **76**, 733–746.
[29] Lewis, R. M. and Torczon, V. J., 1996, Rank ordering and positive bases in pattern search algorithms. Technical Report 96-71, ICASE NASA Langley Research Center, VA, USA.
[30] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V., 1994, *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing* (Cambridge: MIT Press).
[31] Bortz, D. M. and Kelley, C. T., 1998, The simplex gradient and noisy optimization problems. In: Borggaard, J. T., Burns, J., Cliff, E., and Schreck, S. (Eds.), *Computational Methods in Optimal Design and Control*, 77–90 (Boston: Birkhauser).
[32] Kelley, C. T., 1999, Detection and remediation of stagnation in the nelder-mead algorithm using a sufficient decrease condition. *SIAM J. Optim.*, **10**, 43–55.
[33] Hunt, B. R., Lipsman, R. L., and Rosenberg, J. M., 2001, *A Guide to MATLAB: for Beginners and Experienced Users* (Cambridge: Cambridge University Press).
[34] Pawletta, S., *MATLAB DP Toolbox*. Available online at: http://www-at.e-technik.uni-rostock.de/rg_ac/dp/, (2001).
[35] Bongartz, I., Conn, A. R., Gould, N., and Toint, P. L., 1995, CUTE: Constrained and unconstrained testing environment. *ACM Trans. Math. Software*, **21**, 123–160.

[36] Moré, J. J., Garbow, B. S., and Hillstrom, K. E., 1981, Testing unconstrained optimization software. *ACM Trans. Math. Software*, **7**, 17–41.