Asynchronous optimisation with the use of a cascade search algorithm

F. Cecelja, A. Kokossis, D. Du, S. Yang

* Centre for Process & Information Systems Engineering, University of Surrey, Guildford GU2 7XH, United Kingdom

b School of Chemical Engineering, National Technical University of Athens, Athens GR-15780, Greece

A R T I C L E   I N F O

Article history:
Received 3 October 2013
Accepted 14 February 2014
Available online xxx

Keywords:
Markov processes
Asynchronous optimisation
Parallel and distributed computing

A B S T R A C T

This paper introduces the development of an asynchronous approach coupled with a cascade optimisation algorithm. The approach incorporates concepts of asynchronous Markov processes and introduces a search process that is benefiting from distributed computing infrastructures. The algorithm uses concepts of partitions and pools to store intermediate solutions and corresponding objectives. Population inflations are performed periodically to ensure that Markov processes, still independent and asynchronous, make arbitrary use of intermediate solutions. Tested against complex optimisation problems and in comparison with commonly used Tabu Search, the asynchronous cascade algorithm demonstrates a significant potential in distributed operations with favourable comparisons drawn against synchronous and quasi-asynchronous versions of conventional algorithms.

© 2014 Elsevier Ltd. All rights reserved.

1. Introduction

Along with deterministic optimisation, stochastic search based optimisation often referred to as stochastic optimisation (Fouskasis & Draper, 2002), has found many applications in solving complex engineering problems because it overcomes problems associated with non-linearities and discrete variables (Cavin, Fisher, Glover, & Hungerbuhler, 2004; Jayaraman, Kulkarni, Karkale, & Shelokar, 2000). It is based on statistically random probabilistic driven search which guarantees full exploration of search space and exclusion of local optima (Kokossis, Linke, & Yang, 2011). The advantages and disadvantages of stochastic optimisation have been reviewed by numerous authors (Fouskasis & Draper, 2002; Gosavi, 2003; Jayaraman et al., 2000). Yet, in order to ensure convergence and to address search inefficiencies associates with randomness, various strategies have been developed to get use of past history of solutions and to find and favourise the most promising search direction. These have been formalised in a number of optimisation algorithms: perhaps the best known are Simulated Annealing (SA) (Kirkpatrick, Gelat, & Vecchi, 1983), Tabu Search (TS) (Glover, 1989), Genetic Algorithm (GA) (Goldberg, Deb, & Clark, 1992) and Ant Colony Optimisation (ACO) (Dorigo, DiCaro, & Gambardella, 1999). Still, stochastic optimisation generally suffers from slow convergence.

Many contributions were made to overcome slow convergence of stochastic optimisation, which include (i) introducing the level of intelligence in the form of exploiting knowledge about the application (Labrador-Darder, Cecelja, Kokossis, & Linke, 2009; Nandi et al., 2004) or capturing and managing knowledge from the past history of solutions to guide the search (Kokossis, Cecelja, Labrador-Darder, & Linke, 2008), (ii) parallelising or distributing the search process (Cantu-Paz & Goldberg, 2000; Leite & Topping, 1999; Talbi, Hafidi, & Geib, 1998), and (iii) improving existing or developing new algorithms to better accommodate parallel processing by enabling full asynchronous search and knowledge acquisition (Du, Cecelja, & Kokossis, 2011; Kokossis et al., 2008, 2011; Labrador-Darder et al., 2009).

Parallelising the search efforts is perhaps the most commonly reported approach of improving convergence of stochastic algorithms. It exploits advances in computing infrastructure where parallel computing resources are readily available in the form of grid networks or cloud computing across the Internet. Parallel versions of ACO (Reimann, Doerner, & Hartl, 2004; Yang & Zhuang, 2010), GA (Cantu-Paz & Goldberg, 2000; Lim, Ong, Jin, Sendhoff, & Lee, 2007), TS (Borfeldt, Gehring, & Mack, 2003; Cordeau & Maischberger, 2012; Crainic, Toulouse, & Gendreau, 1996; Talbi et al., 1998) and SA (Leite & Topping, 1999; Peierls & Deng, 1998) have been reported. The most common approach is sequential and/or synchronous mode where multiple threads of algorithms are executed sequentially or in parallel exploring different regions with varying degree of collaboration between them (Fouskasis & Draper, 2002; Peierls & Deng, 1998). In order to minimise redundant search, that is to minimise repeated or overlapped moves,
Notation

\( A_{n_{p},t} \) \hspace{1cm} domain of \( D_{n_{p},t}^{h} \)
\( D_{n_{p},t}^{h} \) \hspace{1cm} cascade of \( n_{p} \) pools at time instance \( t \)
\( f \) \hspace{1cm} objective function of an optimisation problem
\( f_{\text{max}} \) \hspace{1cm} maximum \( f \) in \( P_{j,t} \) at time instance \( t \)
\( f_{\text{min}} \) \hspace{1cm} minimum \( f \) in \( P_{j,t} \) at time instance \( t \)
\( F \) \hspace{1cm} superset of numerical functions including the objective function \( f \)
\( F_{\text{max}} \) \hspace{1cm} maximum objective value of the population available at time instance \( t \)
\( F_{\text{min}} \) \hspace{1cm} minimum objective value of the population available at time instance \( t \)
\( g \) \hspace{1cm} a numerical function in \( F \), normally refers to inequality constraints of the optimisation problem
\( g_{l}^{\text{lim}} \) \hspace{1cm} Markov process at time instance \( t \)
\( G \) \hspace{1cm} set of disjoint partitions in \( S \)
\( G_{j,t} \) \hspace{1cm} partition \( j \) at time instance \( t \)
\( G_{n_{p}} \) \hspace{1cm} the lowest partition associated with temperature \( T_{n_{p}} \)
\( h \) \hspace{1cm} a numerical function in \( F \), normally refers to equality constraints of the optimisation problem
\( h_{i,j} \) \hspace{1cm} numerical function over \( G_{i,j} \)
\( h_{j}^{t} \) \hspace{1cm} thread, a software programme to be executed independently
\( h_{s}^{\text{sm}} \) \hspace{1cm} measure of object \((C,P_{j})\)
\( i \) \hspace{1cm} index of points (solutions) in partition (pool) \( j \)
\( j \) \hspace{1cm} index of partitions (pools) from top to bottom, \( j = 1, 2, \ldots, n_{p} \)
\( L \) \hspace{1cm} length of Markov process
\( M_{j} \) \hspace{1cm} number of available points in \( G_{j,t} \)
\( n_{l} \) \hspace{1cm} number of consecutive iterations
\( n_{M} \) \hspace{1cm} size of Markov space (number of Markov processes running in parallel)
\( n_{T} \) \hspace{1cm} number of threads to be executed independently
\( n_{T}_{G_{j}} \) \hspace{1cm} purpose defined integer numbers
\( P_{1} \) \hspace{1cm} the highest pool associated with the lowest quality solutions
\( P_{j,t} \) \hspace{1cm} pool \( j \) at time instance \( t \)
\( P_{n_{p}} \) \hspace{1cm} the lowest pool associated with the highest quality solutions
\( Q_{j,t} \) \hspace{1cm} population of partition (pool) \( G_{j,t} \) at time instance \( t \)
\( Q_{\text{lim}}^{n_{M}} \) \hspace{1cm} Markov space of \( n_{M} \) Markov processes
\( R \) \hspace{1cm} random number distributed uniformly in \([0,1]\) \( \in \mathbb{R} \)
\( S \) \hspace{1cm} feasible region of an optimisation problem
\( s_{i,j,t} \) \hspace{1cm} point \( i \) in partition \( j \) at time instance \( t \)
\( T_{1} \) \hspace{1cm} temperature associated with the highest pool \( P_{1} \)
\( T_{j} \) \hspace{1cm} parameter associated with pool \( P_{j} \) called temperature
\( T_{n_{p}} \) \hspace{1cm} temperature associated with the lowest pool \( P_{n_{p}} \)
\( W_{\text{thr}} \) \hspace{1cm} space of \( n_{T} \) threads
\( \theta \) \hspace{1cm} real number defining the cooling schedule that is distribution of temperatures between \( T_{1} \) and \( T_{n_{p}} \)
\( \varepsilon \) \hspace{1cm} a small number in \([0,1]\) \( \in \mathbb{R} \)

The development of a fully distributed stochastic optimisation algorithms capable of parallel processing without the need for synchronisation still remains a challenge. Cascading algorithm (CA) has been proposed as an approach to break down optimisation threads and Markov chain and hence inherently allowing for parallel execution (Kokossis et al., 2011). This algorithm stores intermediate solutions into pools (and partitions). Similarly to SA, the number of pools is equivalent to temperatures. Pools are ordered according to the quality of solutions (objective values) forming a cascade with the properties, the norm of population and deviation, which can be controlled. The CA search operates iteratively by choosing a candidate point which is accepted/rejected by the probability controlled by the temperature. The cascade requirements reflect the quality of solutions which are verified upon arrival of new points, the process known as inflection. This property of CA enables independent generation of solutions and serves as the base for full asynchronos operation.

Building upon the CA approach, this paper proposes a distributed asynchronous operation of CA which is achieved by implementing asynchronous Markov processes to independently generate new points, and dynamic inflection to minimise redundant moves. More precisely, the process of dynamic inflection of population assures that new Markov processes, although executed independently, benefit from the most recent population of solutions without the need for synchronisation with other processes running in parallel. The whole process is formalised in the form of an Asynchronous Cascade Algorithm. The proposed algorithm is implemented on the Globus upperware with Grid Superscalar used to dispatch computational tasks for parallel computing. To differentiate and report benefits from distributed computing and asynchronos operations, a set of computationally expensive problems has been selected. The comparisons are drawn against optimisation algorithms which include conventional and parallel implementations of SA and TS.

The paper addresses a general formulation for the optimisation problem in the form of:

\[
\begin{align*}
\min & \quad f(x,y) \\
\text{s.t.} & \quad h(x,y) = 0 \\
& \quad g(x,y) \leq 0 \\
& \quad x \in X = [0,1], y \in \{0,1\}^{n}
\end{align*}
\]

where \((x,y)\) accounts for the domain and \((h,g)\) for the problem constraints.

2. Concepts and definitions

2.1 Basic concepts of cascading

The principle of cascading allows for grouping solutions by their quality (Kokossis et al., 2011). The feasible region \( S \) of the optimisation problem \((1)\) is

\[
S = \{ (x,y) | h(x,y) = 0 \land g(x,y) \leq 0, x \in X, y \in Y \}
\]

with the range \( C \) of \( f \)

\[
C = \{ f(x,y) | y(x,y) \in S \}
\]

Let \( G \) be a set of disjoined partitions in \( S \)

\[
G = \bigcup G_{j} \subseteq S
\]

where partition \( G_{j} \) is ordered set of finite elements in \( S \)

\[
G_{j} = \{ s_{1,j}, s_{2,j}, \ldots, s_{M_{j,j}} \}, \quad s_{i,j} \in S, \quad i = 1, 2, \ldots, M_{j}, \quad j = 1.2, \ldots, n_{p}
\]
and where \( n_p \) is the total number of partitions and \( M_j \) is the population size of partition \( G_j \). Defining \( F \) as a set of numerical functions, then for each \( h \in F \), including the objective function \( f \) in (1), every member of feasible region \( s_{ij} \in S \) can be mapped as

\[
f_j : G_j \rightarrow f(G_j) = (f(s_{ij}))^M_{i=1} = (f_i^M)_{i=1}^{M}
\]  

(6)

If \( \exists G_j \in G \) and if \( P_j \) is the range of (6), then \( P_j \) is the pool\(^1\) of \( G \). If, in addition, (6) is observed in more orderly way as

\[
f' : G_j \rightarrow f'(G_j) = (f'(s_{ij}))^M_{i=1} = (f'(s_{ij}))\forall i < k
\]  

then, if \( \exists G_j \subset G \) and if \( P_j' \) is the range of (7), \( P_j \) is an ordered pool of \( G \). Mapping between partitions \( G_j \) in \( G \) and pools \( P_j \) in \( C \) is illustrated in Fig. 1.

As observed from (6) and (7), \( G_j \) is the algebra of \( G \) and \( P_j \) is the algebra of \( C \). Objects \( G_j \) and \( (C, P_j) \) are measurable spaces. \( Q_j \) is the population of \( G_j \) iff \( Q_j \) is the rank of \( G_j (P_j) \):

\[
Q_j = \text{rank}(G_j) = \text{rank}(P_j)
\]  

(8)

Let a h-metric \( h_j \) be defined over \( P_j \) as mapping from \( \mathbb{M}_j \rightarrow \mathbb{R} \), so that

\[
h_j : P_j \rightarrow h_j(p_j) = (f_i^M)_{i=1}^{M}
\]  

(9)

then we can define a measure of the object function \( (C, P_j) \) as

\[
h_j^{\text{in}}(P_j) = h_j^{\text{in}} = \sum_{i=1}^{M} f(s_{ij})
\]  

(10)

as demonstrated in Fig. 1.

Let \( D_{n_p}^h \) be a combination of \( P_j \) ordered by \( h_j^{\text{in}} \) so that

\[
D_{n_p}^h = (P_1, P_2, \ldots, P_{n_p}), h_j^{\text{in}} > h_k^{\text{in}}\forall j > k
\]  

(11)

If \( \exists h \in F \) so that \( D_{n_p}^h \) follows (11), then the combination \( D_{n_p}^h \) of \( n_p \) pools \( P_j \) is called a cascade (a cascade \( D_{3}^3 \) with three pools is shown in Fig. 1).

By observing (11) it is apparent that the domain \( A_{n_p} \) of cascade \( D_{n_p}^h \) is defined as

\[
A_{n_p} = \cup_{j=1}^{n_p} (u_j)^{M}_{i=1}
\]  

(12)

Let \( A'_{n_p} \) be the domain of \( D_{n_p}^h, D_{n_p}^h' \) is defined as inflection of \( D_{n_p}^h \) iff

\[
A'_{n_p} = A_{n_p}
\]  

(13)

Expanding partitions is performed by Markov process

\[
g_t^h : S \rightarrow S
\]  

(14)

over a sequence of time periods \( t \) with the length of a Markov process \( L \). Let

\[
G_{j,t} = \{s_{i,j,1}, s_{i,j,2}, \ldots, s_{i,j,t}\}
\]  

(15)

so that

\[
g_t^h(s_{i,j,k}) = \{s_{i,j,t+1}, s_{i,j,t+2}, \ldots, s_{i,j,t+t}\} \in S,
\]

(16)

and \( G_{j,t+1} \) given by

\[
G_{j,t+1} = G_{j,t} \cup g_t^h(s_{i,j,k})
\]  

(17)

with \( (s_{i,j,t+1}, s_{i,j,t+2}, \ldots, s_{i,j,t+t}) \) observing Markov properties, as shown in Fig. 2 for the first two steps of the partition expansion process. Then the sequence of pools \( P_{j,t+1} \) can be defined over \( G_{j,t+1} \) as

\[
P_{j,t+1} = P_{j,t} \cup \{f(s_{i,j,t+1}), f(s_{i,j,t+2}), \ldots, f(s_{i,j,t+t})\}
\]  

(18)

2.2. Asynchronous cascading

Following (16), each Markov process is employed to generate points \( s_{i,j,k} \). The quality of points is measured by their objective values \( f(s_{i,j,k}) \) which are stored in the pools they are generated from. Each pool \( P_j \) is associated with a global parameter, the pool temperature \( T_j \), which accounts for the level of uncertainty, as is the case of temperature with SA. The pools are arranged in ascending order along \( T_j \). For \( T_j \) and \( T_{np} \) fixed, the individual pool temperatures \( T_j \) are calculated as

\[
T_j = \left( T_1 - T_{np} \right) \frac{1}{1 - n_p^0} + \left( T_1 - T_{np} \right) \frac{1 - T_j}{1 - n_p^0}
\]  

(19)

Here, \( \theta \) is a real number defining the pool distribution along \( T_j \), the process known as the cooling schedule \( \) (Kirkpatrick et al., 1983): (i) for \( \theta > 1 \) the cooling schedule is strictly concave favouring high quality solutions, (ii) for \( \theta < 1 \) the cooling schedule is strictly convex favouring low quality solutions, and (iii) for \( \theta = 1 \) the cooling schedule is strictly linear.

New points \( s_{i,j,t+1} \) are generated from existing \( s_{i,j,t} \) through Markov processes, which, follow the Metropolis acceptance criterion (Foukakis & Draper, 2002) based on the probability \( p_{r,t+1} \)

\[
p_{r,t+1} = P(s_{i,j,t+1} | s_{i,j,t}) = \min \left( 1, \exp \left( \frac{f(s_{i,j,t}) - f(s_{i,j,t+1})}{T_j} \right) \right)
\]  

(20)

---

\(^1\) The full definition of mathematical terms are provided, along with the Glossary at the end of the paper, in Kokossis et al. (2011).
and with new points accepted according to the rule

\[ s_{i,j,t+1} = \begin{cases} 
  s_{i,j,t+1} & \text{if } R \geq R \\
  s_{i,j,t} & \text{if } R < R
\end{cases} \tag{21} \]

where \( R \) is the random number distributed uniformly in \([0,1] \in \mathbb{R}\). As from \((20)\), the probability of accepting new points is dictated by the pool temperature \( T_j \); for \( T_j \) large (small) the Markov process follows high (low) probability to accept new points.

With asynchronous Markov process, new points are generated in parallel and independently of each other. By observing \((9)\) and for \((G, G_j)\) being measurable spaces, let \( g^i_{t,M}(s_{i,j,k}) \) be Markov processes over \( G_{i,t} \) mapping from \( G \to G \) as

\[ g^L_{t,M}(s_{i,j,k})_{i=1}^{nM} = [s_{i,j,t+1,1}, s_{i,j,t+2,1}, \ldots, s_{i,j,t+L_1}]_{i=1}^{nM} \geq \sum_{i=1}^{nM} M_j \]

\[ k = 1, 2, \ldots, t \]

and hence forming a Markov space \( Q^M \) of \( nM \) processes.

Let \( W^M \) be a space of \( nT \) threads \( h_j, j = 1, 2, \ldots, nT \) accounting for the completion of each \( g^L_{t,M}(s_{i,j,k}) \) as shown in Fig. 3. For \( nM = nT \), the implementation accounts for a bijective process. For \( nM = 1 \) the entire process becomes sequential.

Observing \((17)\) and following the acceptance \((21)\) yields

\[ G_{i,t+1} = G_{i,t} \cup g^L_{t,M}(s_{i,j,k}) \tag{23} \]

and the sequence of pools \( P_{j,t} \) over \( G_{i,t} \) is now

\[ P_{j,t} = (f(s_{i,j,t+1}), f(s_{i,j,t+2}), \ldots, f(s_{i,j,t+L}))_{i=1}^{nM} \tag{24} \]

New inflections are developed periodically following \((13)\) and to conform to \((11)\). They are based on minimum \( F^\min \) and maximum \( F^\max \) objective values of each current population (at each given time instance \( t \)). Let \( P_j \) has a boundary of objectives values \( F^\min \) and \( F^\max \) which depend on the selected temperature range \( T_{np} \) to \( T_1 \) and which are evenly distributed across pools as

\[ \frac{T_j - T_{j+1}}{T_1 - T_{np}} = \frac{F^\max - F^\min}{F^\max - F^\min} \tag{25} \]

It is apparent that there is total of \( n_T + 1 \) boundaries and from \((24)\) and \((25)\) they are

\[ f^\min_{j,t} = \frac{(F^\max - F^\min) \times (T_i - T_j) + F^\min}{T_1 - T_{np}} \tag{26} \]

\[ f^\max_{j,t} = \frac{(F^\max - F^\min) \times (T_{j+1} - T_j) + F^\min}{T_1 - T_{np}} \tag{27} \]

so that the points \( f(s_{i,j,k}) \) are distributed among the pools following

\[ f^\max_{j,t} \geq f(s_{i,j,k}) \geq f^\min_{j,t} \tag{28} \]

as illustrated in Fig. 4.

As with CA \((\text{Kokossis et al., 2014})\), termination criteria are based on the population size, the progress of the optimisation process and population features as:

a) \textbf{Population size}: the population \( Q_j \) of either \( G_{i,t} \) or \( D_{np,t} \) can be observed and the termination can be based on the following:

- The algorithm does not terminate if \( Q_j < Q_{\text{min}} \) assuring that premature conversion does not take place;
- The algorithm terminates if \( Q_j \geq Q_{\text{max}} \) assuring convergence.

Selection of \( Q_{\text{min}} \) and \( Q_{\text{max}} \) is application dependent.

b) \textbf{Optimisation progress}: the process terminates when there is no further improvement in the quality of the solutions. Let \( 0 \leq \varepsilon \leq 1, \varepsilon \in \mathbb{R} \) and \( nT \) being an integer, the process terminates when

\[ \left\| f^\min_{t+\varepsilon} - f^\min_{t} \right\| \leq \varepsilon \tag{29} \]

for \( n_T \) consecutive iterations.

c) \textbf{Population feature}: population features are observed at pool level. Let \( \sigma_t \) be standard deviation in a pool, normally the pool with the highest quality of solutions, and \( n_{GT} \) an integer. The algorithm terminates if

\[ \left\| \sigma_t - \sigma_{t+\varepsilon} \right\| \leq \varepsilon \tag{30} \]

for \( n_{GT} \) consecutive iterations.
2.3. Asynchronous cascading algorithm

In its asynchronous cascading implementation, the cascade optimisation algorithm uses \( n_p \) partitions \( G_{j,t} \) and pools \( P_{t,j} \), as shown in Fig. 5. The set of \( n_p \) pools \( P_{t,j} \) forms the cascade \( D_{np}^t \). Given the top \( (T_1) \) and bottom \( (T_{np}) \) temperatures, each pool is associated with a pool temperature \( T_t \) following (19). A set of \( n_M \) Markov processes \( \mathbf{G}_{j,t}^{nM} \) increases the population in \( G_{j,t} \) and \( P_{t,j} \). The proposed approach is employing a bijective \( (n_M = n_T) \) implementation with \( n_M \) Markov processes, each running as a separate thread. Each of \( n_T \) threads is executed by a separate worker\(^2\) \( W_t \).

The asynchronous cascading algorithm is outlined in Fig. 6 and takes the following form:

1. Select the set of at least \( n_M \) initial points \( S_0 \) and respective objectives \( f(S_0) \) and place them into \( G_{1,t} \) and \( P_{1,t} \), respectively;
2. Check if the cascade follows (11) and if not, develop an inflection by redistributing population according to (26) and (27) so to conform to (11) and (13);
3. Map \( n_T \) points in \( G_{1,t} \) into new points following (22). Check acceptance (21) and place new points and their objectives into \( G_{L,t} \) and \( P_{1,t} \), respectively. Otherwise iterate with Step 2;
4. Check the termination criteria and iterate with Step 2.

In the full asynchronous implementation, the inflection in Step 2 is performed after each new point and respective objective value are placed into \( G_{1,t} \) and \( P_{1,t} \). Other schedules of inflection are possible, e.g. after certain number of new points has been generated. This could save some execution time but could be at the expense of redundant points. The process of generating new points in Step 3 takes place as soon as a thread is free for execution, thus there is no wait-in-time involved.

Points \( G_{L,t} \) to initialise new Markov process in Step 3 are selected from randomly selected partitions \( G_{j,t} \). Let randomly selected partition \( G_{r,t} \) include all points from higher partitions such

\[
G_{r,t} = \cup_{j=r}^{np} G_{j,t}
\]

\(^2\) The term worker refers to a computer/server which is a part of the computer grid and capable of executing given thread independently, yet sharing data with other servers/computers as necessary.

![Fig. 5. Asynchronous CA execution.](image)

![Fig. 6. Asynchronous CA algorithm flow diagram.](image)

![Fig. 7. Implementations of asynchronous CA pools to feature high quality solutions.](image)
1 to be selected is \( p^1_t = 0.002 \), whereas the probability for a solution from pool 5 to be selected is 5 times higher as \( p^5_t = 0.01 \).

The optimisation converges when one or more termination criteria are met. In the present implementation, the criteria include progress, population size limits and population distributions. Considering population sizes, the optimisation terminates if the population in the highest pool \( P_{\text{hi}} \) does not change within predefined number of consecutive inflections \( n_{\text{pre}} \). Considering population distributions, the optimisation terminates if the standard deviation of the objective values in the highest pool \( P_{\text{hi}} \) falls below predefined threshold \( \sigma_t \) for a predefined number of consecutive inflections \( n_{\text{ST}} \).

### 2.4. Parallel implementation of Tabu Search

Although cascading could lead to a full asynchronous search, a certain degree of asynchronous elements can be achieved with conventional stochastic algorithms. Tabu Search (TS) was selected as a method widely established, tested and reasonably sophisticated to compare with CA and so that to demonstrate possible benefits. The parallel version of TS has also been reported (Crainic et al., 1996), but in its implementation it still involves elements of synchronisation after certain number of iterations and to minimise redundant moves.

Tabu Search (TS) has been explained elsewhere (Cordeau & Laporte, 2005; Crainic et al., 1996; Glover, 1989; Linke & Kokossis, 2003b). It is in many ways ordinary local or neighbourhood search which proceeds from one solution to another until the chosen termination criteria is satisfied (Glover & Laguna, 2011). As such, TS repeatedly performs transitions from current to new states or solutions in its neighbourhood. In optimisation terms, TS follows a simple descend and permits moves to neighbouring solutions which improve current objective value. The search terminates when no improving solution can be found. In order to minimise redundant moves, that is to minimise revisiting previously visited solutions and/or cycling in local optimum, a TS algorithm has introduced the Tabu list in the form of a short term memory. Moves revisiting the solutions are not allowed. In fact, the moves on the Tabu list are the reverse or opposite of the moves recently applied and the Tabu list is updated with the reverse moves. Hence, starting from the current point \( s_j \), a set of \( n_{\text{EN}} \) sub-neighbour solutions \( \{ s_{ij} \}_{i=1}^{n_{\text{EN}}} \) are explored (Fig. 9). If a solution in the set satisfies the acceptance criterion, that is if its objective value \( f(s) \), \( i \in \{1, 2, \ldots, n_{\text{EN}}\} \) is better than the current best solution \( f^*(s_k) \), and if it is not on the Tabu list, it will replace \( f^*(s_k) \).

The introduction of the Tabu list in the TS algorithm is often seen as a radical approach in view of the possibility that the moves on the list might still attain potentially attractive and previously unvisited solutions. The aspiration criterion overrides the Tabu status if its aspiration \( a(s, m) \) marks better than a pre-set threshold value \( A(s, m) \):

\[
a(s, m) < A(s, m) \quad i = 1, \ldots, n_{\text{EN}}
\]  \hspace{1cm} (33)

Tabu lists are performed provided that at least one (or some specified number) of conditions (33) are satisfied (Glover, Taillard, & Werra, 1993).

A long term memory, commonly called the frequency long term memory (FLM) (Glover et al., 1993; Linke & Kokossis, 2003b), is introduced to enable learning from the past search experience. It records the frequency with which individual moves have been performed over the search history and hence provide bias towards promising areas, the intensification, as much as towards areas that have not been visited before, the diversification. In consequence, intensification generates neighbouring solutions by grafting together good solutions and encouraging search into areas with historically good solutions and returning attractive regions to search them more thoroughly. The diversification, on the other hand, encourages search process to visit unvisited regions and to generate solutions that differ in various significant ways from those seen before.

A commonly used solution acceptance criterion for TS is comparison of the current solution with the best and respectively taking better (Glover, 1989).

Similarly to our proposed algorithm, TS uses several termination criteria which include (Glover, 1989; Glover et al., 1993; Linke & Kokossis, 2003b):

a) **Population size**: the optimisation terminates after pre-defined number of iterations,

b) **Optimisation progress**: the optimisation terminates after a predefined number of iterations without improvement of the best solutions \( f^*(s_k) \) is reached, and

c) **Population feature**: the optimisation terminates after reaching predefined threshold solution value.

The whole process is depicted in Fig. 8.

In its commonly used implementation, a dynamic neighbourhood size is searched from predefined point (Wang, Quan, & Xu, 1999). In the present implementation, we use Markov process to generate new population by exploring a fixed size \( n_{\text{EN}} \) neighbourhood \( \{ s_{ij} \}_{i=1}^{n_{\text{EN}}} \) from the initial instance \( s_0 \) (Fig. 9). The best new solution \( f(s_{ij}) \) is compared to the current best solution \( f^*(s_k) \) and, if better and not on the Tabu list, \( f^*(s_k) \) is replaced by \( f(s_{ij}) \) and Tabu list and FLM are updated accordingly. For moves rendered Tabu, the corresponding solution is accepted, if it satisfies the aspiration criteria (33). The neighbourhood size is further increased by repeating the whole process for a fixed number of iterations \( n_t \), which form a slot. After one slot is completed, the current best solution \( f^*(s_{ij}) \) is selected as the initial solution for the next slot. The optimisation process continues until the termination criteria is met: in the present implementation the optimisation terminates when...
predefined number of slots have been executed or no better solution has been found after predefined number of slots, whichever is reached first.

Intensification aims to explore neighbourhood \( \{s_{i,l}\}_{i=1}^{n_{EN}} \) more thoroughly and it is considered by performing moves which are recorded in FLM with a high success rate. Similarly, diversification encourages the search process to explore low rate regions. In the present implementation, both intensification \( I_i \) and diversification \( D_i \) processes are implemented as separate slots executed after regular slots in the Markov process, as shown in Fig. 10. All slots, including \( I_i \) and \( D_i \), have the same neighbourhood size \( n_{EN} \) and the same number of iterations \( n_t \). The three concomitant solutions, \( f^*(s_{jj}) \) from the regular slot, \( f_i^*(s_{i,l}) \) from intensification and \( f_i^*(s_{i,l}) \) from diversification are compared and the best, \( f_i^*(s_{i,l}) = \max(f^*(s_{jj}), f_i^*(s_{i,l}), f_i^*(s_{i,l})) \), is taken to serve as the initial solution for the search in the next slot. Also, the best current solution is replaced as \( f_i^*(s_{i,l}) = \max(f^*(s_{jj}), f_i^*(s_{i,l}), f_i^*(s_{i,l})) \). The Tabu list and FLM are updated accordingly.

Parallel execution and corresponding thread allocation of multiple TS model is shown in Fig. 11, where \( n_c \) separate chain searches are processed by \( n_T \) different workers. The number of chains \( n_c \) is not necessarily the same as the number of workers \( n_T \); more common choice is \( n_c \gg n_T \). Each of the \( n \) regular slots in the chain are normally followed by one intensification slot \( I_k \) and one diversification slot \( D_k \), the ‘1+2’ model. In the ‘1+4’ model we use 2 intensification and 2 diversification slots following each of \( n \) regular slots in a chain.

In the present implementation, each chain starts search from different randomly chosen initial point \( s_{ij}, i=2, \ldots, n_c \). In order to follow TS concept, the requirement that each individual chain, not necessarily as a single thread, is executed sequentially should not be violated.

From the practical perspective, a thread should be formed to ensure that the execution time \( t_e \) of the thread executed on the slowest worker is greater than the computational overheads \( t_{com} + t_p \), that is \( t_e > t_{com} + t_p \) (Badia et al., 2003), where \( t_{com} \) is the communication time to the workers, and \( t_p \) is the thread preparation time. On the other side, the thread should be sufficiently small in terms of number of slots to minimise redundant search which might occur because of the lack of synchronisation between threads and despite of existence of Tabu list and FLM. Naturally smallest allocation is for each thread involving individual neighbouring search; this however would require a strict synchronisation performed at the end of each iteration to determine \( f\left(s_{ij}\right) \).

In the present implementation we adopted a quasi-asynchronous approach where all threads are of the same size and have the same ‘1+2’ or ‘1+4’ format, as illustrated in Fig. 11. All slots have the same neighbourhood size \( n_{EN} \) and the same number of iterations \( n_t \). A separate memory, the processing memory, is used to keep...
the track of execution hence ensuring sequential execution of each individual chain.

2.5. Parallelisation

Grid computing, and more recently introduced cloud computing, have become a very important research and development area because they offer a possibility of parallel execution of threads, tasks and even jobs, and hence significant benefits from computing resources widely available. Speeding up the execution is perhaps the most obvious benefit when dealing with complex application algorithms. Most of the grid environments, however, require complex programming skill to deal with the workflow in the form of specifying the workflow dependencies usually in non-imperative languages. GRIDS superscalar which runs on the top of Globus toolkit is one of the exceptions where writing an application is “as simple as programming a sequential programme running on a single processor with hardware resources totally transparent”, and it was selected for the implementation in this work (Badia et al., 2003).

Conceptually, the GRID superscalar framework takes the master–worker form with master computer having the process managing role to distribute threads to each worker and monitoring the execution, while the workers are local or remote processors executing the threads (Fig. 12).

The behaviour of GRID application depends on data dependences between threads, which are the input and output parameters of each thread. Threads without dependences can be executed immediately. Example of a GRIDS application with C threads is given in Fig. 13, together with the sample of the programme resembling an ordinary sequential programme. Here, THi is the ith iteration of kth thread: in each thread iterations THi and THj cannot start before the iteration THi is finished, and the iteration THi cannot start before iterations THi and THi are completed. Each thread uses the same form of input data so they can start execution independently.

For the implementation of the asynchronous cascade algorithms presented in this paper, two types of threads were introduced: the server thread, or the optimisation server, and the Markov process thread, or the thread (Fig. 14). The optimisation server runs on the master and performs the following functions:

1. Distribution of threads to all nT workers Wj together with the initial solutions sij;
2. Receiving new solutions from the workers and verifying them according to the acceptance criteria in (21) for CA or (33) for TS;
3. Distributing and storing the solutions and respective objective values into the pools/partitions for CA and populating Tabu list and frequency long term memory (FLM) for TS;
4. Verifying the optimisation termination criteria.

The Markov process thread executes asynchronous Markov process for CA, and 1 regular and one intensification and one diversification slot (‘1 + 2’ format) or, one regular and two intensification and two diversification slots (‘1 + 4’ format) for TS on selected worker Wj to generate new points from initially selected sij.

3. Experimental evaluation

3.1. Illustrative examples

Two examples are selected to demonstrate reasonable complexity and they are sufficiently demanding to justify asynchronous implementations and the need for parallelisation. They are referred to as Experiments 1 and 2. Their formulations account for non-convex, non-linear problems with discrete elements that model the selection of engineering units (Mehta & Kokossis, 1997).
Experiment 1 involves structural choices whereas Experiment 2 involves very nonlinear terms. Exact formulations of the used experiments are presented in Appendix, whereas respective local searches are detailed elsewhere (Linke & Kokossis, 2003a; Mehta, 1998; Mehta and Kokossis, 1997). The importance of the problem behind the formulations used in the experiments is discussed extensively in the literature (Ashley & Linke, 2004; Giorno & Drioli, 2000; Kokossis & Floudas, 1990; Lei & Jorgensen, 2001; Marcoulaki & Kokossis, 1999). The next section concentrates on the aspects of the asynchronous implementation focusing on the asynchronous cascade algorithm. We also experiment with SA algorithm obtained from CA by eliminating inflections. The last section compares the asynchronous cascade algorithm with quasi-asynchronous implementations highlighting, in contrast to the new approach, diminishing benefits as the number of workers increase.

3.2. Experiments with asynchronous cascade algorithm

Observing Section 2, the asynchronous CA structure is defined by the number of pools (partitions) \( n_p \), the highest \( T_1 \) and the lowest \( T_n \) temperatures, length of Markov process \( L \) and the cooling schedule \( \theta \). Effects of all of these parameters have been tested using set-up in Experiments 1 and 2 in more than 10 different configurations each.

Experimental results demonstrate that by increasing the number of pools \( n_p \), the quality of the solutions improves dramatically and quickly achieves excellent results as shown in Fig. 15a. For linear and convex cooling schedules with \( \theta \leq 1 \) the improvement is better for \( n_p \geq 70 \), whereas for concave cooling schedule with \( \theta > 1 \) better improvement relates to \( n_p \geq 100 \). Results in Fig. 15a are very similar to those obtained in Experiment 2, which is in many ways regarded as a very complex optimisation problem. Hence \( n_p = 100 \) appears to be a good choice and was used in most experiments presented in this paper.

Cooling schedule determines the distribution of pools. As illustrated in Fig. 15b, for a strictly convex schedule (\( \theta < 1 \)) the optimisation process has better convergence as the latter is measured by a lower standard deviation of the solutions in the highest pool \( p_{n_p} \). This is at the expense of significantly higher population \( Q \) (\( Q \) is the total population in all \( n_p \) pools) and longer time to converge. This type of cooling schedule favours lower quality solutions and hence focuses more on exploring regions not previously visited. For linear and concave schedule (\( \theta \geq 1 \)) the process converges faster but

---

Appendix is presented only in the electronic version of the paper.

---

Please cite this article in press as: Cecelja, F., et al. Asynchronous optimisation with the use of a cascade search algorithm. Computers and Chemical Engineering (2014), http://dx.doi.org/10.1016/j.compchemeng.2014.02.009
The effects of CA optimisation structure (Experiment 1, \( n_T = 1 \) (Pentium® 4 3.00 GHz), \( L = 10, T_1 = 100, T_{np} = 0.01 \)).

Table 1

<table>
<thead>
<tr>
<th>Experiment</th>
<th>((t_{com} + t_{p}) (s))</th>
<th>(t_c (s))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experiment 1</td>
<td>22</td>
<td>65</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>22</td>
<td>384.45</td>
</tr>
</tbody>
</table>

Table 2

<table>
<thead>
<tr>
<th>Worker</th>
<th>Type</th>
<th>(t_p (s))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(W_1, W_2)</td>
<td>Intel Pentium®, 3.00 GHz</td>
<td>182.5</td>
</tr>
<tr>
<td>(W_3, W_4)</td>
<td>Intel Pentium III, 700 MHz</td>
<td>384.45</td>
</tr>
</tbody>
</table>

Please cite this article in press as: Cecelja, F., et al. Asynchronous optimisation with the use of a cascade search algorithm. *Computers and Chemical Engineering* (2014). http://dx.doi.org/10.1016/j.compchemeng.2014.02.009
In all experiments with asynchronous CA we used the combination of population size and population feature as the termination criteria. The optimisation terminates if the population size in the lowest pool $P_n$, does not increase for 20 consecutive redistributions of the population, or if the standard deviation of the objective values in the highest pool $P_h$, is below 0.05 for Experiment 2 and below 0.02 for Experiment 1 for 15 consecutive redistributions, whichever comes first.

3.3. Experiments with parallel Tabu Search

The TS structure is defined by the neighbourhood size $n_{bn}$, depth of search in terms of number of iterations in each slot $n_i$ and introduction of intensification and diversification searches. The effects of these parameters were studied using both Experiments 1 and 2 in more than 10 different configurations each.

Neighbourhood size $n_{bn}$ is seen as a compromise between the resolution of the search and the size of redundant moves: larger the $n_{bn}$, more thorough search is ensured but also higher the probability of revisiting the points because not all solution within one iteration are registered on the Tabu list. It has been established here that by increasing the neighbourhood size $n_{bn}$ the TS algorithm converges faster. This benefit, however, vanishes for neighbourhood size of $n_{bn} \geq 10$ when the overall population size becomes dominant. An example with Experiment 2 is illustrated in Fig. 18 where the increase in the neighbourhood size from $n_{bn} = 6$ to 7 improves convergence measured by the overall execution time (Fig. 18a) but significantly increases the population size $Q$ measured by normalised time $t_e/Q$ (Fig. 18b). Note that population size $Q$ here refers to all solutions used for compliance with Tabu list, that is the total number of solutions generated by the threads.

It is apparent from Fig. 18 that due to effect of Amdahl's law the improvement in performance of TS virtually vanishes with number of workers $n_p > 10$

Depth of the search is defined by the number of iterations $n_i$ in a slot. It has been established that depth of search does not have significant effects on the performance as long as the number of iteration is big enough to ensure sufficient population for moderating the granularity of the search, and also small enough to avoid high rate of redundant search. For the applications presented in this paper, $n_i = 4 \div 10$ was found to be a good compromise between the two, which was confirmed with many more applications we experimented with.

The introduction of the intensification and diversification does not have immediate effect on the performance, as shown in Fig. 19. In Fig. 19 the term ‘single model’ refers to optimisation without any diversification or intensification slots, whereas the term ‘multiple model’ refers to optimisation with one intensification and 1 diversification slot introduced after each regular slot, that is ‘1+2’ model. Although with modest contribution, introduction of intensification and diversification searches in the form of ‘1+2’ model helps in reaching the optimal solution by avoiding local optima but at the expense of increasing the population size (Fig. 19b). Introducing additional diversification and intensification slots, such as ‘1+4’ model in Fig. 20, further increases population without obvious beneficial effects on performance.

Diagrams in Figs. 18–20 demonstrate that parallelising speeds up the execution proportionally to the number of computers employed. As with asynchronous CA, this improvement vanishes when the computing overhead time $\sum_{i=1}^{n_p}(t_{i,com} + t_{i,p})$ becomes comparable to the thread execution time $t_e$ of the slowest worker (Eq. (34)). For more complex implementation, i.e. with
Fig. 18. Effect of neighbourhood size $n_{EN}$ on TS search (Experiment 2 experiment, ‘1 + 2’ model, $n_{I} = 5$).

Fig. 19. Effects of intensification and diversification on TS optimisation (Experiment 2, ‘1 + 2’ model, $n_{I} = 5$).

Fig. 20. Results for ‘1 + 4’ model of TS optimisation (Experiment 2, $n_{I} = 5, n_{EN} = 7$).

Fig. 21. Effects of different power of workers on TS optimisation (Experiment 2, $n_{I} = 5, n_{EN} = 7, ‘1 + 2’$ model).
larger neighbourhood sizes or by introducing intensification and diversification, it is apparent that higher number of workers is needed to reach maximum performance, as demonstrated in Figs. 19 and 20. Introduction of slower workers, however, deteriorates the performance significantly as the whole operation is dominantly synchronous, as illustrated in Fig. 21 where ‘6+0’ configuration refers to experiment with all workers having the same computational power (Intel® Pentium® 4 3.0 GHz) and ‘3+3’ configuration refers to experiment with 3 fast workers (Intel® Pentium® 4 3.0 GHz) and 3 much slower workers (Intel Pentium III 700 MHz).

3.4. Comparative analysis

Although easier to implement, it is apparent that TS optimisation algorithm generally suffers from slower convergence and higher population size Q than SA and asynchronous CA algorithms, as shown in Table 3. On the other side, SA and asynchronous CA algorithms have much more options to adapt to the optimisation problem and to benefit from the past history of solutions. These include choice of cooling schedule, as well as grouping and qualitative discrimination of population by the instrument of pools and partitions. In consequence, SA and CA algorithms achieve much better convergence with smaller population. Also, these two algorithms show similar convergence behaviour because they share the same search methodology, acceptance criteria, as well as the termination criteria. All three algorithms, however, reach the same quality of results as demonstrated by the best final solution $p_{\text{max}}$ in Table 3. Note that results presented in Table 3 are all from a single computer implementation ($n_f = 1$).

In parallel implementation TS demonstrates better convergence performance than SA algorithm (Fig. 22b). However, much higher population size diminishes this advantage making TS and SA algorithm very similar in practical terms (Fig. 22a). Introduction of cascading and inflection, however, shows obvious advantages of parallel optimisation evident in that asynchronous CA algorithm performs significantly better convergence in terms of computing time (Fig. 22a) and lower population size (Fig. 22b), as compared to SA and TS algorithms. In our experiments savings achieved with asynchronous CA algorithms is measured in hours. Much more significant advantage of CA algorithm, however, is that it can accommodate higher level of parallelisation; its fully asynchronous capability reduces idling time of workers, which in turn reduces effects of Amdahl's law allowing for a higher number of workers to be beneficially employed, as illustrated in Fig. 22a. Adding to this the introduction of inflection and cascading which enable asynchronous CA algorithm to converge with lower population size further differentiate asynchronous CA algorithm from others, as shown in Fig. 22b.

All three optimisation methods, CA and asynchronous CA, SA and TS, have provided virtually the same solutions for the reactor network sequence, concentration of the output product and the total reactor volume, as shown in Table 4. These results are the best of the group of results obtained in the course of experimenting with the performance of the three algorithms and which are estimated to be the real optimum of the respective problems, not the local optima (Fig. 23).

### Table 3. Performance of TS, SA and CA algorithms.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>$p_{\text{max}}$</th>
<th>$p_{\text{max}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS</td>
<td>171</td>
<td>3.681</td>
</tr>
<tr>
<td>CA</td>
<td>142</td>
<td>3.691</td>
</tr>
<tr>
<td>SA</td>
<td>142</td>
<td>3.691</td>
</tr>
</tbody>
</table>

*Tabu Search configuration: ‘1 + 2’ model, $n_i = 5$, $n_{\text{SA}} = 6$, $n_f = 1$.

*Simulated annealing configuration: $L = 10$, $n_y = 100$, $\theta = -2$, $T_1 = 100$, $T_y = 0.01$.

*Asynchronous CA algorithm: $L = 10$, $n_y = 100$, $\theta = -2$, $n_i = 1$, $T_1 = 100$, $T_y = 0.01$. 

### Table 4. Reactor network solutions.

<table>
<thead>
<tr>
<th>Model</th>
<th>Concentration ratio</th>
<th>Total volume (l)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CA</td>
<td>SA</td>
</tr>
<tr>
<td>Experiment 1</td>
<td>3.641</td>
<td>3.641</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>5.152</td>
<td>5.153</td>
</tr>
</tbody>
</table>
4. Conclusion

The full asynchronous version of stochastic optimisation in the form of the cascade algorithm which aims to employ available distributed computing resources is presented. The algorithm introduces partitions and solution pools arranged in cascades together with asynchronous Markov search to generate new points. The solutions in pools are inflected periodically to further minimise redundant search. Similarly to SA, each pool is associated with the temperature decreasing towards the highest pool and which helps with inflection process. The synchronous version of CA resembles SA algorithm. In addition, a parallel version of Tabu Search is also presented. Individual threads are composed to minimise the need for mutual synchronisation which resembles quasi-asynchronous operation. Two complex experiments of reactor network synthesis are used to assess the performance of CA in asynchronous operation implemented on GRIDS superscalar environment. Exhaustive experimentation and comparison with TS and SA shows superiority of asynchronous operation of CA in benefitting from distributed computing resources and respectively speeding up the optimisation process. Future papers will focus on acquiring knowledge in the process of optimisation and introduction of knowledge models to further improve quality and convergence of stochastic optimisation.

Appendix A. Supplementary data

Supplementary data associated with this article can be found, in the online version, at http://dx.doi.org/10.1016/j.compchemeng.2014.02.009.

References


G Model
CACE-4893; No. of Pages 14

F. Cecelja et al. / Computers and Chemical Engineering xxx (2014) xxx–xxx

Glossary

Pool: for a function \( f_i : G_i \rightarrow (\mathbb{C}G_i) \), a set \( P_i \) is a pool of \( G \) if \( \mathbb{C}G_i \subset G \) so that \( P_i \) is the range of the function \( f_i \).

Cascade: cascade is a combination of pools ordered by some pool properties, usually by the measure of respective object function.

\( \sigma \)-Algebra: \( \sigma \)-algebra (usually signed as \( \sum \)) is a collection of sets which satisfy certain properties and with the main use to define measure.

Inflection: inflection is a process of redistribution of the population between pools along some properties of the population.

Please cite this article in press as: Cecelja, F., et al. Asynchronous optimisation with the use of a cascade search algorithm. Computers and Chemical Engineering (2014), http://dx.doi.org/10.1016/j.compchemeng.2014.02.009