An efficient fault-tolerant communication algorithm for population-based metaheuristics

Amanda S. Dufek
National Laboratory for Scientific Computing
Petrópolis-RJ, Brazil
Lawrence Berkeley National Laboratory
Berkeley-CA, USA
asdufek@lbl.gov

Douglas A. Augusto
Oswaldo Cruz Foundation
Rio de Janeiro-RJ, Brazil
daa@fiocruz.br

Helio J. C. Barbosa
National Laboratory for Scientific Computing
Petrópolis-RJ, Brazil
hcbm@lncc.br

Pedro L. S. Dias
University of São Paulo – IAG
São Paulo-SP, Brazil
pedro.dias@iag.usp.br

Jack R. Deslippe
Lawrence Berkeley National Laboratory
Berkeley-CA, USA
jrdeslippe@lbl.gov

ABSTRACT

Parallel and distributed computing systems have been seeing rapid growth in the number of processing cores as progress on single-core performance has stagnated. The larger the system, the greater the challenge for application scalability and system stability. Aiming at addressing both challenges in the context of distributed metaheuristic optimization algorithms, in this work, we propose a scalable and fault-tolerant peer-to-peer communication algorithm tailored for population-based metaheuristics. In the algorithm, messages exchanging are carried out by multiple threads asynchronously in background and the minimal algorithm's overhead can be entirely hidden by overlapping communication with computation. Results from controlled benchmarks corroborate the efficiency of the algorithm and also hint that thread oversubscription can further improve scalability thanks to the high degree of idleness of communication operations. The proposed algorithm contributes to the important yet not sufficiently explored performance aspects of distributed metaheuristics.

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**; • **Theory of computation** → *Optimization with randomized search heuristics*; • **Computer systems organization** → *Peer-to-peer architectures*; *Fault-tolerant network topologies*;

KEYWORDS

Metaheuristics, Distributed communication algorithm, Asynchronous, Scalable, Fault-tolerant

GECCO '21 Companion, July 10–14, 2021, Lille, France

ACM ISBN 978-1-4503-8351-6/21/07.
https://doi.org/10.1145/3449726.3463144

ACM Reference Format:
Amanda S. Dufek, Douglas A. Augusto, Helio J. C. Barbosa, Pedro L. S. Dias, and Jack R. Deslippe. 2021. An efficient fault-tolerant communication algorithm for population-based metaheuristics. In *2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3449726.3463144

1 INTRODUCTION

Metaheuristics define general-purpose high-level strategies to guide the development of heuristic optimization algorithms [27, 29] and are usually capable of performing global search. They can be based on a single solution — S-metaheuristics — or on a population of solutions — P-metaheuristics. Well-known examples of metaheuristics include the family of Evolutionary Algorithms, Simulated Annealing, Ant Colony Optimization, Particle Swarm Optimization, Tabu Search, Variable Neighborhood Search, Iterated Local Search, among others [29].

Upon solving a problem, particularly complex real-world problems, a common approach to improve the efficiency of metaheuristics is to distribute the search among algorithm instances in a *cooperative* way, with each instance possibly having different parameters (homogeneous) or even being different metaheuristics (heterogeneous) [29]. This decomposition, known as algorithmic-level parallel model, is conceptually simple and relatively easy to implement atop existing metaheuristics; however, its main benefits entail the improvement of *effectiveness*, *scalability* and *robustness* [5, 6, 29, 32].

In this context, the design and implementation of the layer that performs the communication for exchanging messages among algorithm instances is a major determinant of the overall performance of algorithmic-level parallel metaheuristics, especially concerning scalability and robustness. Here is where our proposed communication algorithm tailored for P-metaheuristics comes in.

Unlike other parallel models in which some computationally costly procedures in P-metaheuristics (e.g. evaluation phase) are parallelized, the algorithmic-level parallel model accelerates the

execution of a P-metaheuristic by dividing its population of $|P|$ candidate solutions into p independent runs of the algorithm, each one with a population size of $|P|/p$; that is, a single and costly run of the algorithm is replaced by p faster ones executed in a parallel cooperative way. Here we will concentrate on the algorithmic-level parallel model, although P-metaheuristics can be further decomposed into multiple complementary parallel models, thereby achieving a high degree of parallelism and scalability [4, 10, 26, 29].

There have been several researchers interested in the development of parallel and distributed models for P-metaheuristics. Most of their research focus on studying how the model's parameter settings and implementation strategies can contribute to population diversity, quality of solutions, and the time required to obtain them [3, 21, 22]. However, implementation and infrastructure details of the communication models as well as their behavior in terms of efficiency, scalability and robustness have not been much studied by researchers.

One of the major issues concerning the design of a robust parallel and distributed system is its ability to continue operating properly in case of a fault on a processor or on the communication channel. In fact, that is a particularly serious problem for grid computing systems, which are inherently unreliable due to their volatility. Indeed, in non-dedicated distributed computing environment failures are inevitable since volunteers' machines can be turned off at any moment and even when running they are usually available only while they are in an idle state [14]. Unfortunately, a large number of publications in this area have not taken into account the problems arising from the emergence of faults that may break down all algorithm instances. Message Passing Interface (MPI) is an example of a communication interface that does not provide an inherent fault-tolerance support by default.[1] For such cases, there are some recovery techniques from which a state of the running instances prior to the failure is restored [13, 25, 30, 31]. However, such techniques have important shortcomings. In addition to losing the failed messages and the progress since the last checkpoint, the periodical storage of state data makes the approaches much more complex and computationally costly. On the other hand, when using an inherently fault-tolerant communication interface — like the socket-based peer-to-peer model [28] — connection failures may lead to lost messages but the remaining instances continue to run seamlessly; also, there is evidence that message exchange failures by themselves do not adversely affect the quality of solutions in algorithmic-level parallel models [16].

Another issue of paramount importance regarding the design and efficiency of communication models is the definition of the message exchange pattern across the processes. Synchronous parallel strategies are easier to implement than their asynchronous counterparts, but an obvious drawback of them is the time spent by the processes waiting for each other so that all communication is carried out at the same time. That leads to overall performance degradation with respect to scalability, and an increase in communication overhead with the number of processes [1, 11]. In contrast, the processes in asynchronous parallel strategies communicate independently, which is clearly advantageous when it comes to performance.

In this work we present an asynchronous and fault-tolerant peer-to-peer communication algorithm based on sockets for P-metaheuristics, with emphasis on the effectiveness, scalability and robustness, as well as on the design and implementation aspects. Other research on asynchronous and fault-tolerant approaches to some specific P-metaheuristics can be found in the literature [12, 17–19, 33, 34]. The most important differences between them and the current paper are: (i) it can be applied to any population-based metaheuristic; (ii) it minimizes the algorithm's overhead by communication-computing overlapping; (iii) the parallel asynchronous execution of multiple threads for messages exchanging takes place in background; (iv) the effective time spent on synchronous blocks is negligible; (v) it is optimized for modern heterogeneous system architectures; (vi) it maximizes the effective use of computing power; (vii) messages are received continuously as soon as they arrive and integrated into the population in the next iteration; and (viii) the mechanism of sending messages is fine-grained, being capable of carrying out the communication at every iteration.

2 PEER-TO-PEER COMMUNICATION ALGORITHM

2.1 Peer-to-peer P-metaheuristics

In the cooperative algorithmic-level parallel model, many independent runs of P-metaheuristics are launched simultaneously in a parallel cooperative way, each of them assigned to a different local or remote processor, in order to solve a given optimization problem. It is also known as island model when dealing with a particular class of P-metaheuristic called evolutionary algorithms [8, 9]. In terms of implementation, it basically consists of a P-metaheuristic coupled with a communication algorithm for exchanging messages, whose implementation varies according to the parallel and distributed architecture (clusters, networks of workstations, grids) and programming environment (Message Passing Interface, Parallel Virtual Machine, Sockets) [29].

A general scheme of the algorithm proposed here in pseudo-code is outlined in Algorithm 1, and it will be explained in the following paragraphs.

Algorithm 1: Peer-to-peer P-metaheuristics (adapted from [29])

```
1  server.start(); [async]                          // peer is ready to receive messages
   t ← 0;
3  P_t ← initialize(); P'_t ← ∅;                    // initialization of the population
   evaluate(P_t);                                   // evaluate population
   while stopping criteria not met do
6      P'_t ← P'_t ∪ generate(P_t);                 // generate next population
7      e ← evaluate(P'_t); [async]
8      send(P_t);          // send selected solutions from P_t to remote peers
9      P'_{t+1} ← receive();     // transfer the received solutions to P'_{t+1}
       waitFor(e);               // wait until evaluate(P'_t) has finished
11     P_{t+1} ← select(P_t ∪ P'_t);                // form next population
       t ← t + 1;
   return the best solutions found
```

[1] To be fair, MPI can be made more fault-tolerant by adjusting its behavior regarding error handling [15], but this approach is not straightforward and rarely adopted by MPI programs.

P-metaheuristics begin with the generation of an initial population of candidate solutions, P_0 (line 3). The role of the population is to hold candidate solutions to a problem. Each iteration of a P-metaheuristic includes three main procedures: (i) the generation of a new population of solutions, P'_t (line 6); (ii) the evaluation of solutions according to some objective function, which assigns a quality measure to the solutions (line 7); and (iii) the replacement of the current population, P_t, by another one, P_{t+1}, composed of solutions selected from the current, P_t, and the new, P'_t, populations (line 11). This process iterates until a stopping criterion is satisfied [29].

We propose a peer-to-peer communication algorithm to incorporate the algorithmic-level parallelism into P-metaheuristics by adding three procedures: (i) the continuous receipt of solutions sent by remote peers (line 1 — Algorithm 2 in Section 2.4); (ii) the sending of selected solutions from the previous population, P_t, to each remote peer (line 8 — Algorithm 4 in Section 2.5); and (iii) the transfer of received solutions to the next population, P'_{t+1} (line 9 — Algorithm 3 in Section 2.4). In case P'_t in line 6 already has some solutions due to the execution of line 9 in the previous iteration, the `generate()` function will be responsible for generating just the remaining solutions into the next population; otherwise, it will fully generate the next population. Note that each peer assumes both server and client roles and there is no central agent. Hereafter, the term *client* will be used to refer to a given peer when it is sending messages to (connecting to) remote peers, and the term *server* will be used to refer to a peer when it is receiving messages (listening to connections) from remote peers. Beware not to confuse the terminology used with client-server communication model.

Given that the evaluation phase (line 7) is typically the most computationally costly procedure in P-metaheuristics, it runs concurrently with both the `send()` and `receive()` functions (lines 8 and 9). That is, the algorithm overlaps computation and communication, which in practice fully hides the communication effort. It is worth noticing that the solutions sent to remote peers belong to the previous population P_t, i.e. they do not belong to the population P'_t, whose solutions are undergoing the process of evaluation. Furthermore, after the multi-threaded server starts (line 1), whenever a client connects to it a thread is automatically spawned in background to receive the message. The `send()` and `receive()` functions in turn are executed every iteration. Hereafter, the algorithm resulting from the union of P-metaheuristic and the peer-to-peer communication algorithm will be referred to as peer-to-peer P-metaheuristics (in short, P2P-metaheuristics).

Therefore, a P2P-metaheuristic minimizes the impact of the communication operations on the total execution time while maximizing the effective use of computing power. Although we focus on population-based metaheuristics, the communication algorithm can also be applied to metaheuristics based on a single solution with minor adaptations in line 6. For instance, we can replace it by $P'_t \leftarrow$ `generate(`P_t,P'_t`)`, in which a new single solution is obtained from two previous ones.

2.2 Illustrative scheme

This section aims at presenting a general view of the execution of the described P2P-metaheuristics through illustrative schemes to facilitate the understanding of its overall functioning.

Figure 1 shows a hypothetical communication topology between three processes: P_1, P_2 and P_3. For the given topology, the processes

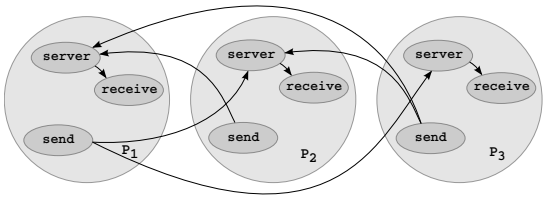


Figure 1: Hypothetical communication topology with three processes: P_1, P_2 and P_3. Each process P is represented by a multi-threaded *server* and by the `send()` and `receive()` functions.

P_1 and P_3 send messages to the other two processes, while P_2 sends messages only to P_1. Therefore, P_1 and P_2 receive messages from the other two processes, while P_3 receives messages only from P_1. The connection between *server* and `receive()` describes the transfer of received solutions to the population. It is worth mentioning that there are no restrictions with regard to the communication topology, i.e. it does not necessarily have to follow a well-known graph structure such as ring, star and mesh topologies.

Figure 2 depicts an illustrative execution timeline of the computation and communication tasks of P2P-metaheuristic executed by P_1; they are: (i) the incoming message processing (server's `run()` and `receive()`); (ii) the generation of a new population of solutions (`generate()`); (iii) the evaluation of solutions according to some objective function (`evaluate()`); (iv) the outgoing message processing (`send()`); and (v) the formation of a new population of solutions (`select()`). The asynchronous message exchanges between P_1 and the other two processes, P_2 and P_3, are assigned to sending threads, T_{s_1} and T_{s_2}, and receiving threads, T_{r_1} and T_{r_2}, running in background. The server's `run()` is automatically ex-

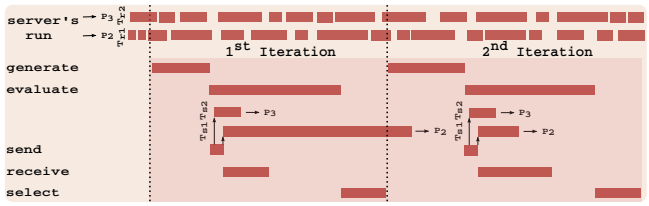


Figure 2: Illustrative execution timeline visualization of the computation and communication tasks of a P2P-metaheuristic executed by process P_1. Vertical dashed lines indicate the beginning of an iteration. Note that only the first two iterations were reproduced here.

ecuted by a receiving thread whenever P_2 and P_3 connect to P_1. The scheme of Figure 2 highlights a particular case where the completion of sending a message to the process P_2, initialized in the first iteration by the thread T_{s_1}, occurs only in the next iteration. It shows that the message exchange operations do not interfere with the execution of the P-metaheuristic, both running concurrently. Moreover, the communication overhead relative to the `send()` and `receive()` functions is reduced or even completely hidden by overlapping them with `evaluate()`.

2.3 Peer-to-peer communication model

The communication among P-metaheuristics follows the peer-to-peer model based on sockets [28], and they exchange some information during the iterative search through message passing within a node or via a network (cluster, network of workstations, grid). Each P-metaheuristic instance has its own socket and multi-threads responsible for sending and receiving messages, and assumes both server and client roles. Each socket in turn is associated with an Internet Protocol (IP) address and a port number.

The most important features of the proposed communication algorithm are as follows:

(i) **Efficiency and scalability:** the algorithm aims at maximizing the overlap between computation and communication tasks as well as the effective use of computing power, and minimizing the synchronous operations.
(ii) **Fault tolerance:** given that we are using an inherently fault-tolerant communication interface, failures on processors or on the communication channels are ignored in the sense that incomplete messages are lost but the remaining processes keep operating correctly. Furthermore, if/when the failed processes come back, their respective exchanges of messages are automatically reestablished.
(iii) **Asynchronous:** there is no synchronization barrier among the algorithm instances preceding the communication operations. The message exchanges are executed by multiple threads in a parallel asynchronous way, running in background.
(iv) **Topology:** the communication topology can follow any graph structure. Moreover, the initial topology can be dynamically modified by adding or removing processes or connections. It is also possible to modify the parameters of P-metaheuristics on-the-fly.
(v) **Message exchanged:** although we have been systematically referring to messages as *solutions*, there is nothing in the proposed algorithm that prevents exchanging other types of information, such as parameters, search memory, similarity measures, among others [2, 20].
(vi) **Architecture:** there are no restrictions with regard to the parallel and distributed architectures neither the latency, bandwidth and reliability of the communication channel. However, as the algorithm is based on sockets, which by design are not optimized for low-latency high-throughput networks, these networks will not contribute substantially performance-wise.

Therefore, the peer-to-peer communication model based on sockets seems to be the most suitable proposal for incorporating the algorithmic-level parallelism into P-metaheuristics when the message exchanging follows the pattern shown in Algorithm 1. In this context, the socket-based approach brings an obvious advantage over MPI: it works satisfactorily on every network architecture, even on slow and unreliable ones such as grid computing. Despite that, MPI would be definitely preferred for communication-intensive applications that demand high-performance interconnects (i.e. low-latency and high-bandwidth), and may afford access to dedicated and fully reliable network.

2.4 Incoming message processing

Whenever a client connects to a server, the server's `run()` function, shown in Algorithm 2, is automatically executed by a thread, with the maximum number of receiving threads defined *a priori*. Using the `rcvMessage()` function (line 5), the multi-threaded server receives the solution sent by the client, and temporarily stores it into buffer vector — called buf — in an asynchronous way. The buffer vector consists of a vector of pointers to characters of size buf_{max_size}. It is shared by all receiving threads. The set of statements within an indented block of code by the `lock` directive — a critical section — is executed by only one thread at time, i.e. the receiving threads perform in a synchronous way. The access control of receiving threads to critical sections follows the FIFO system (*First in, first out*).

Algorithm 2: Server's `run()`

```
1 if writeQueue.empty() then return();
  lock
      if writeQueue.empty() then return();
      position ← writeQueue.pop();
5 rcvMessage(buf[position]);
  lock
      readQueue.push(position);
```

Let *writeQueue* and *readQueue* be queues shared by all receiving threads, and composed of available positions of the buffer vector for writing and reading operations, respectively. Thus, an element of *writeQueue* represents an available position of the buffer vector for temporarily storing the solution coming from a client; while an element of *readQueue* is a position of the buffer vector whose solution is ready to be included in the server's population. The queue *readQueue* is initially empty while the queue *writeQueue* contains all the buf_{max_size} positions of the buffer vector.

According to line 1 in Algorithm 2, if the *writeQueue* is empty, the incoming message processing by the thread in question is interrupted. However, it can be easily replaced by a `while` loop that will wait until the buffer vector is available for a write operation.

At the end of each iteration of the P-metaheuristic, `receive()`, shown in Algorithm 3, is executed until a stopping criterion is satisfied: a maximum number of arriving solutions buf_{max_size} in the population is reached or *readQueue* is empty (line 1). The solution, whose position in the buffer vector is given by the first element of *readQueue* (line 3), is transferred to the server's population by `copyToPopulation()` (line 4). Once the solution transference is finished, its position in the buffer vector is released (line 7) in order to store a new solution by the server's `run()` function.

Note that the read and write operations are exclusive, thus *writeQueue* and *readQueue* have no elements in common, and the element removed from a queue is inserted into the other queue after the completion of the tasks: `rcvMessage()` (line 5 of Algorithm 2) and `copyToPopulation()` (line 4 of Algorithm 3).

Since the server's `run()` and `receive()` functions are independent and the first one is executed by multiple threads, the `pop()` and `push()` operations on the two queues are synchronous by using the `lock` directive. As a result, a queue cannot be assigned to two or more threads simultaneously. The server's `run()` function and

Algorithm 3: `receive()`

```
1 while not readQueue.empty() and buf_size < buf_max_size do
      lock
3     position ← readQueue.pop();
4     copyToPopulation(buf[position]);
      buf_size ← buf_size + 1;
      lock
7     writeQueue.push(position);
```

`receive()` aim at minimizing the number of `locks` as well as the number of statements within the critical sections.

2.5 Outgoing message processing

Let $k+1$ be the number of processes and K be the set containing all the k servers, $s_1, s_2, \ldots, s_k$, available for each client. At first, a client can connect to any server. However, depending on the communication topology adopted, a client will connect to a subset of K — called S — where 2^k is the number of possible subsets. The communication between a client and its subset of servers S is done concurrently, with the number of sending threads equal to the size of S, given by $|S|$. Each pair client-server is associated with a sending frequency — called $frequency$ — that represents the probability that the client will send a solution to the server. For instance, if the sending frequency of a given pair client-server is equal to one, then it has 100% probability.

At the end of each iteration of the P-metaheuristic, the process of sending a solution from the client's population — called *solution* — to the server s_i, $i = 1, \ldots, |S|$, is initiated by the `send()` function, shown in Algorithm 4.

Algorithm 4: `send()`

```
1 for i ← 1 to |S| do
2     if [probabilistically] frequency[i] then
3         if thread[i].isRunning() then continue();
4         thread[i].solution ← selection();
5         thread[i].run(); [async]
```

At each iteration i of the for loop (line 1), if the conditional statement is true (line 2), the client's `run()` function (line 5) is executed by the thread i that sends *solution*, selected from the client's population (line 4), to the server s_i. The line 3 checks whether the thread i, run in the previous iteration, is still running. If that is the case, the sending of a new solution to the server s_i is interrupted until the previous one ends.

The client's `run()` function, shown in Algorithm 5, contains just one statement. Using the `sndMessage()` function, the client

Algorithm 5: Client's `run()`

```
sndMessage();
```

requests a connection to the server, sends a solution and disconnects. In case the connection between client-server is not established in a predefined period of time, the outgoing message processing by the thread in question is interrupted.

Note that because of the complete loop independence in Algorithm 4, with just a simple OpenMP `parallel for` directive [7] it is possible to parallelize the loop inside the `send()` function (line 1), and so distribute the $|S|$ iterations of the `for` loop among OpenMP threads.

3 BENCHMARKS

The communication behavior of the algorithm presented in the previous section is evaluated in terms of efficiency and scalability on three benchmarks, which carry out a separate analysis of each communication operation as well as both of them working together. The benchmark discussed in Section 3.1 takes into account only the incoming message processing, while the one addressed in Section 3.2 considers solely the outgoing message processing. Thereafter, the entire process of exchanging messages is shown in Section 3.3.

The proposed communication algorithm has been implemented in the C/C++ programming language using the API provided by the portable POCO C++ Libraries [23]. POCO offers a variety of convenient routines for peer-to-peer communication and multi-threading, such as: (i) network socket, (ii) multi-threaded server that listens for connections in background and spawns worker threads accordingly, (iii) thread pools, and (iv) thread synchronization mechanisms.

We introduced the communication algorithm into an existing P-metaheuristic powered by an evolutionary algorithm known as grammatical evolution [24], which searches — by means of principles of natural selection — for symbolic regression and classification models that minimize the prediction error according to a training dataset. The implementation is dubbed *Parallel Program Induction* (`PPI`)[2], and was specifically adapted to the benchmarks in order to make the results general and valid for any P-metaheuristics.

All the experiments were conducted on a single node featuring two 10-core Intel E5-2690v2 CPUs (*Hyper-Threading* was disabled to ensure reliable analysis), totaling 20 physical cores, running a 64-bit Debian GNU/Linux system. The implementation was compiled with GNU GCC 6.3, with optimization flags enabled. Although the benchmarks run on a single node, the latency and bandwidth of a network are simulated therein, making such parameter controllable, which would not be possible in a real-world scenario. A choice of a single node does not impair the experiments since we are assessing the efficiency and scalability of the algorithm in terms of execution time, speedup and receiving rate. We are not worried about experimentally evaluating the algorithm's fault-tolerance ability because (i) the algorithm is fault-tolerant by design; and (ii) the socket-based peer-to-peer communication interface provides an inherent fault-tolerance support by default (see Section 2.3 (ii)).

3.1 Incoming message processing

The benchmark reported in this section simulates the incoming message processing in order to analyze the effects of varying (i) the maximum number of receiving threads and (ii) the execution time of the `rcvMessage()` function (line 5 of Algorithm 2) on the time

[2] `PPI` is a Free Software available at `http://github.com/daaugusto/ppi` [10]

required by the server's `run()` function to receive 1000 solutions, each one of size 1000 bytes. The outgoing message processing is switched off temporarily, allowing us to focus only on the incoming message processing. Although in practice the communication among P-metaheuristics is relatively low, the purpose of the current experiment is to exhaust the receiving threads with an intense flow of messages in order to evaluate the limiting behavior of the server's `run()` and `receive()` functions. The communication topology consists of a central server that is connected to $|S| = 5$ remote clients, as illustrated in Figure 3a. The central server receives messages from

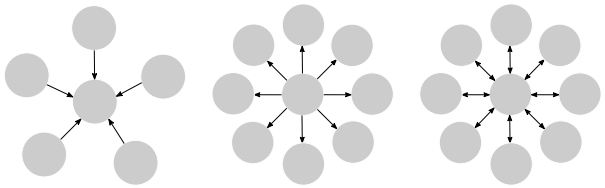


Figure 3: (a) A central server and five remote clients (Section 3.1). (b) A central client and eight remote servers (Section 3.2). (c) A central peer and eight remote peers (Section 3.3).

all remote clients, and every remote client sends messages only to the central server. The maximum number of receiving threads was expressed as a power of 2 ranging from 1 to 64. The different times to execute the `rcvMessage()` function draw an analogy to some possible combinations between latency and bandwidth that together dictate the speed and capacity of transferring data over the local network or Internet as well as network problems. That is achieved by including a `sleep()` function in the scope of the `rcvMessage()` function, whose argument varied from 5 milliseconds to 1 second. For each combination between number of threads and data transfer time, a total of ten independent runs were performed, and the resulting median execution time and speedup over them was used to provide a performance analysis.

Figure 4a-b shows a three-dimensional plot of the median execution time of the server's `run()` and `receive()` as a function of the number of receiving threads and the data transfer time set by the `sleep()` function in the `rcvMessage()` function. It is observed that the total execution time required by the server's `run()` to receive all solutions decreases as the number of threads increases, dropping off rapidly for the domain between 1–16 threads and 500–1000 milliseconds (see Figure 4a). Indeed, the process of incoming messages exhibits a very good scalability, and even scale past the number of physical threads due to the high degree of idleness of communication operations (see Figure 5a). The speedup is near-linear with respect to the number of threads. It means that the impact of the synchronous operations followed by the `lock` directive on the incoming message processing is negligible. In addition, the lower the number of threads, the more pronounced is the slope relative to the data transfer axis, as we can see in Figure 4a. In other words, the total time taken to receive all solutions becomes less dependent on the time spent by a single run as the number of threads increases.

The receiving rate of messages decreases proportionally to the data transfer time over network. For instance, if server's `run()` runtime takes 1 second, just one solution has been received in this time interval by a single thread. However, if server's `run()` runtime is 100 times faster, i.e. it takes 10 milliseconds, it is expected that 100 solutions have been received within 1 second by a single thread, and so on. The server's `run()` function receives messages at a rate that varies from 1 to 196 solutions per second, depending on the data transfer time (see Figure 5b). Given that the receiving rate is worth 196 solutions per second when set up with a `sleep()` time of 5 milliseconds, the time attributed to statements other than `rcvMessage()` is about 0.1 millisecond, being the lock time around 0.04 millisecond. Note that the receiving rate is invariant with the number of threads.

The median execution time of the startup tasks that precede the execution of the server's `run()` function is roughly 35 milliseconds, independently of the number of threads (see Figure 5c [3]). It is worth mentioning that the startup tasks run only once at the beginning of P2P-metaheuristic (line 1 of Algorithm 1), becoming irrelevant with regard to its total execution time.

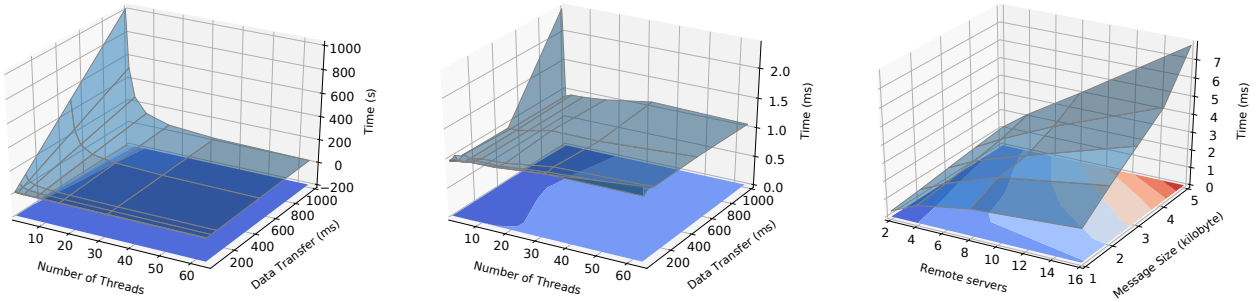
After the multi-threaded server receives all the 1000 solutions of size 1000 bytes asynchronously in background, the `receive()` function takes roughly 1 millisecond transferring them to the population sequentially, reaching 2.4 milliseconds in some runs (see Figure 5d). It is worth mentioning that the server's `run()` continues receiving solutions while `receive()` is running. Therefore, the threads compete for the locks, which adds a slight delay to the end of `receive()`. By consequence, `receive()` takes a slightly longer time as the number of threads increases (see Figures 4b and 5d). However, according to Figure 4b, the maximum median runtime was 2.4 milliseconds when set up with one thread and transfer data time of 1 second. It biased the mean and median from a sample of 10 runs×6 data transfer time relative to one thread, as shown in Figure 5d. Unlike the other configurations, that can be seen as a persistent behavior. Nevertheless, this configuration is unusual in practice, besides being an isolated case. Furthermore, the aforementioned runtime is irrelevant in regard to the server's `run()` runtime, whose values varied from 0.10 to 1005.80 seconds (see Figure 4a).

According to Algorithm 1, `receive()` runs immediately after `send()` launches the sending threads, and both run concurrently with the evaluation of the objective function. Therefore, `send()` and `receive()` do not introduce additional time into the P-metaheuristics runtime when the execution time of `evaluate()` is higher than that obtained by `send()` and `receive()` together. The `send()` runtime will be discussed in the next section.

3.2 Outgoing message processing

The benchmark conducted in this section provides a performance analysis of the outgoing message processing as a function of two parameters: message size and number of remote servers. The incoming message processing is switched off temporarily. The communication topology consists of a central client that is connected to $|S|$ remote servers, as illustrated in Figure 3b for $|S| = 8$. The central client always sends one solution selected from a population of size

[3] A box plot presents a quick sketch of the distribution of a data set. The box is bounded by the upper and lower quartiles, and thus locates the central 50% of the data. The bar inside the box is the median. Open circles represent the sample mean. The whiskers extend from the quartiles to the most extreme value that is within one-half times the distance of the interquartile range away from the quartiles. Data beyond the end of the whiskers are plotted individually [35].



(a) server's `run()` (b) `receive()` (c) `send()`

Figure 4: (a) Median execution time of (a) the server's `run()` to receive 1000 solutions, each one of size 1000 bytes (Section 3.1); (b) `receive()` to transfer them to the population as a function of two parameters: (i) the maximum number of receiving threads and (ii) the data transfer time over network (Section 3.1); and (c) `send()` over ten runs as a function of two parameters: (i) message size and (ii) number of remote servers (Section 3.2). `Send()`'s median runtime does not take into account the `sndMessage()` runtime.

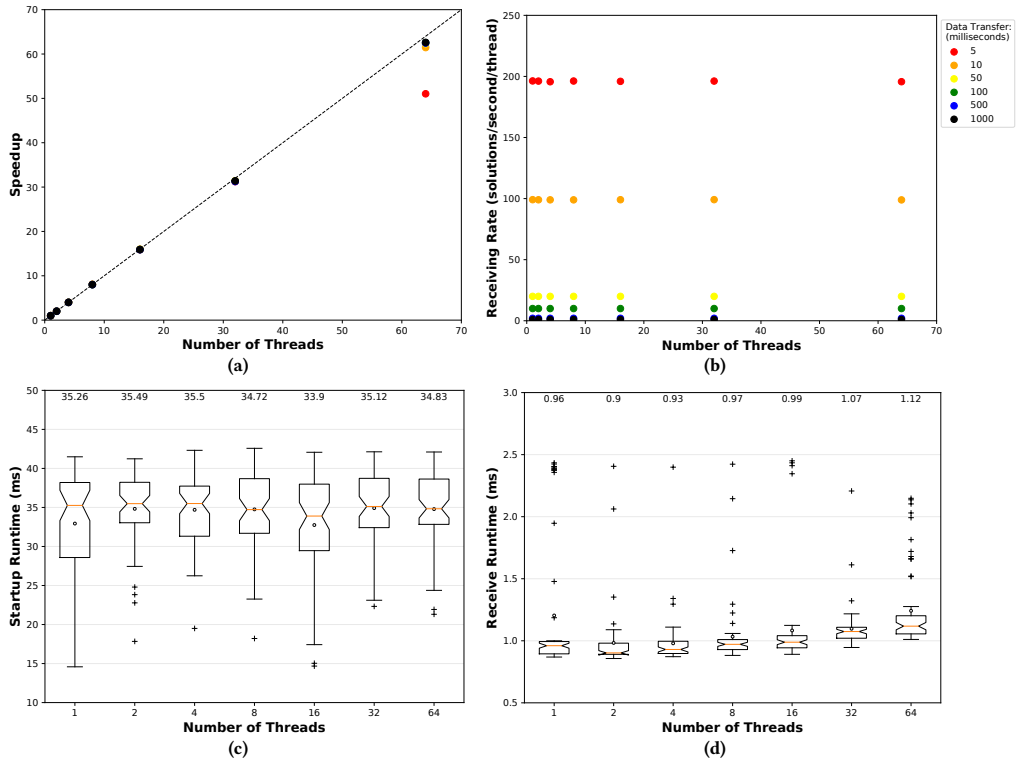


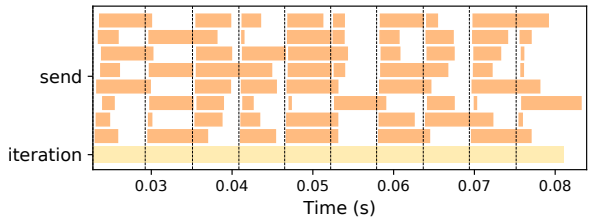
Figure 5: (a) Median speedup and (b) receiving rate corresponding to the server's `run()` as a function of the number of receiving threads for values of data transfer time between 5 and 1000 milliseconds. The dashed black line depicts the ideal linear speedup. Box plots of (c) startup tasks and (d) `receive()` runtime over a sample of 10 runs×6 data transfer time as a function of the number of receiving threads. Median values are displayed at the top of the graph.

1000 to each remote server at each iteration. We measure the total time required by the `send()` function to send messages, with sizes ranging from 1000 to 5000 bytes, to all the $|S|$ remote servers, with $|S| = 2, 4, 8, 16$. Ten independent runs were performed for each combination between message size and number of remote servers. The `send()` runtime does not take into account the execution time of `sndMessage()` (line 5 of Algorithm 4) since it is highly dependent on the speed of transferring data over the local network or Internet. Anyway, it does not interfere with the 3-D surface-shape of Figure 4c.

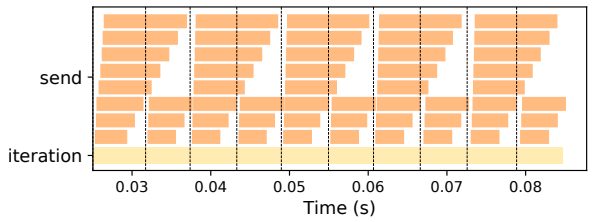
Figure 4c shows a three-dimensional plot of the median execution time of the `send()` function as a function of message size and number of remote servers. As expected, the `send()`'s median runtime increases linearly with the message size at a rate of 0.20–1.41 milliseconds per 1000 bytes, whose values grow with the number of remote servers. Analogously, the linear growth rate of the `send()`

runtime with respect to the number of remote servers varied from 0.10 to 0.48 milliseconds per server, depending on the message size. Given that the `send()` and `evaluate()` functions run concurrently, the `send()` effort is fully hidden when the effective time spent on evaluating the solutions lies above the 3-D surface, whose values ranged from 0.32 to 7.79 milliseconds.

In the next batch of experiments, we simulate sending messages over network using a `sleep()` function in the scope of the `sndMessage()` function. Figure 6 depicts an execution timeline of the iterative search of the P-metaheuristic coupled with the outgoing message processing with eight remote servers. It shows how `sndMessage()` behaves under different amounts of time it takes for a message to travel from the client to the server. In Figure 6a,



(a) Random sending time



(b) Fixed sending time

Figure 6: Execution timeline visualization of the iterative search of P-metaheuristic coupled with the outgoing message processing with eight remote servers: (a) random and (b) fixed sending time. The sending threads are displayed in ascending order of fixed sending time. Vertical dashed lines indicate the beginning of an iteration. The time spent by a single iteration was roughly 6 milliseconds. Note that only the first ten iterations were reproduced here.

at each iteration of the `for` loop in Algorithm 4, the sending time was randomly sampled over the interval from 0 to 10 milliseconds; whereas in Figure 6b a different fixed sending time was assigned to each of the eight sending threads, whose values varied from 3 to 10 milliseconds. As mentioned in the previous section, it draws an analogy to some possible combinations between latency and bandwidth of a network as well as network problems. Note that the completion of sending some messages occurs only in the iteration following that of submission. In that cases, the sending of a new message is interrupted until the previous one ends.

3.3 Incoming + outgoing message processing

In the current benchmark, both the incoming and outgoing message processing are switched on in order to provide a complete view of the P2P-metaheuristic. The experiment was carried out with the following parameters: eight receiving threads, eight sending threads, message size of 1000 bytes, and population of 1000 solutions. The sending time was randomly sampled over the interval from 0 to 10 milliseconds, and the receiving time was set at 5 milliseconds. The communication topology consists of a central peer that is connected to eight remote peers. The central peer receives messages from all remote peers, and sends one solution to each remote peer at each iteration, as illustrated in Figure 3c.

Figure 7 shows an execution timeline of the P2P-metaheuristic with eight threads for each of the incoming and outgoing message processing. It highlights the overlap between computation and

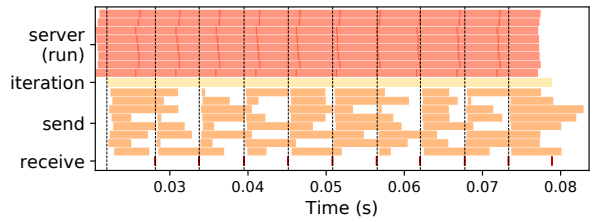


Figure 7: Execution timeline visualization of the P2P-metaheuristic with eight threads for each of the incoming and outgoing message processing. The time spent by a single iteration was roughly 6 milliseconds.

communication tasks, with up to eighteen threads running simultaneously, without interfering with each other. By consequence, the impact of the communication operations on the total execution time of the P-metaheuristic is minimal.

4 CONCLUSIONS

A communication algorithm based on sockets has been incorporated into P-metaheuristics, resulting in what we have termed P2P-metaheuristics. Three main functions make up the communication algorithm: server's `run()`, in which the multi-threaded server listens to and receives the solutions sent by remote peers; `receive()`, the transfer of received solutions to population; and `send()`, the sending of selected solutions from population to remote peers.

P2P-metaheuristics aim at (i) reducing the communication overhead by overlapping communication with computation; (ii) maximizing the use of processing cores available on the system; (iii) minimizing the number of `locks` as well as statements within critical sections on the incoming message processing. Moreover, the communication algorithm is fault-tolerant, asynchronous and orthogonal to topology.

Benchmarks that simulate the incoming and outgoing message processing [4] have shown the efficiency and scalability of the communication algorithm. Given that (i) the effective time spent on synchronous blocks is small enough that it does not degrade performance, (ii) `send()` and `receive()` run concurrently with the evaluation of the objective function, (iii) `evaluate()` runtime is typically greater than a few tens of milliseconds in practice, and (iv) the parallel asynchronous execution of multiple threads for message exchange takes place in the background, we can conclude that the communication algorithm runs almost entirely in the background of

[4] To the best of our knowledge, there is no research on communication costs and multi-threading scalability issues in the context of algorithmic-level parallel P-metaheuristics in order to conduct a comparative analysis based on the three benchmarks presented in Section 3.

a P-metaheuristic. Put differently, the proposed P2P-metaheuristic is in practice as fast as the corresponding P-metaheuristic, even though it continuously communicates with an arbitrary topology of peers whatever the communication reliability.

The proposed P2P-metaheuristic fits well into modern heterogeneous parallel computing, in which the metaheuristic tasks are strategically partitioned among the available compute resources in order to accelerate the execution of the algorithm considerably. For instance, irregular workloads can be assigned to conventional CPUs whereas accelerator devices handle more regular workloads such as the evaluation phase. At the same time, some lightweight communication processes are executed by CPU cores in background.

Finally, we believe that the asynchronous, efficient, scalable, and fault-tolerant communication algorithm as proposed in this work can contribute much to the overall performance of algorithmic-level parallel P-metaheuristics. An immediate follow-up research would be to implement and evaluate the parallelization of the outgoing message processing. Another interesting study would be to evaluate the performance of the algorithm on real-world problems as well as its scalability to thousand of processes.

ACKNOWLEDGMENTS

The authors would like to thank the support provided by CNPq (grants 310778/2013-1, 502836/2014-8 and 300458/2017-7), FAPEMIG (grant APQ-03414-15), EU H2020 Programme and MCTI/RNP–Brazil under the HPC4E Project (grant 689772), and DOE U.S. Department of Energy (DE-AC02-05CH11231).

REFERENCES

[1] Enrique Alba and José Ma Troya. 1999. *An analysis of synchronous and asynchronous parallel distributed genetic algorithms with structured and panmictic Islands.* Springer Berlin Heidelberg, Berlin, Heidelberg, 248–256.

[2] Lourdes Araujo, Juan Julián Merelo Guervós, Carlos Cotta, and Francisco Fernández de Vega. 2008. MultiKulti Algorithm: Migrating the Most Different Genotypes in an Island Model. *CoRR* abs/0806.2843 (2008).

[3] Lourdes Araujo, Juan Julian Merelo, Antonio Mora, and Carlos Cotta. 2009. Genotypic Differences and Migration Policies in an Island Model. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO '09).* ACM, New York, NY, USA, 1331–1338.

[4] D.A. Augusto and H.J.C. Barbosa. 2013. Accelerated Parallel Genetic Programming Tree Evaluation with OpenCL. *J. Parallel Distrib. Comput.* 73, 1 (Jan. 2013), 86–100.

[5] Erick Cantú-Paz and David E. Goldberg. 2000. Efficient parallel genetic algorithms: theory and practice. *Computer Methods in Applied Mechanics and Engineering* 186 (June 2000), 221–238.

[6] Erick Cantú-Paz and David E. Goldberg. 2003. Are Multiple Runs of Genetic Algorithms Better than One?. In *Genetic and Evolutionary Computation - GECCO 2003, Genetic and Evolutionary Computation Conference, Chicago, IL, USA, July 12-16, 2003. Proceedings, Part I.* 801–812.

[7] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. 2001. *Parallel Programming in OpenMP.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[8] J. P. Cohoon, S. U. Hegde, W. N. Martin, and D. Richards. 1987. Punctuated Equilibria: A Parallel Genetic Algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application.* L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 148–154.

[9] James P. Cohoon, Shailesh U. Hegde, Worthy N. Martin, and Dana S. Richards. 1991. Distributed genetic algorithms for the floorplan design problem. *IEEE Trans. on CAD of Integrated Circuits and Systems* 10, 4 (1991), 483–492.

[10] Amanda Sabatini Dufek, Douglas Adriano Augusto, Helio José Corrêa Barbosa, and Pedro Leite da Silva Dias. 2018. *Multi- and Many-Threaded Heterogeneous Parallel Grammatical Evolution.* Springer International Publishing, 219–244.

[11] Francisco Fernández, G. Galeano, and J.A. Gómez. 2002. *Comparing Synchronous and Asynchronous Parallel and Distributed Genetic Programming Models.* Springer Berlin Heidelberg, Berlin, Heidelberg, 326–335.

[12] Gianluigi Folino and Giandomenico Spezzano. 2006. P-CAGE: An Environment for Evolutionary Computation in Peer-to-Peer Systems. In *Genetic Programming*, Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 341–350.

[13] Christian Gagne, Marc Parizeau, and Marc Dubreuil. 2003. Distributed Beagle: An Environment for Parallel and Distributed Evolutionary Computations. In *Proceedings of the 17 th Annual International Symposium on High Performance Computing Systems and Applications (HPCS).*

[14] Daniel Lombraña González, Francisco Fernández de Vega, and Henri Casanova. 2010. Characterizing fault tolerance in genetic programming. *Future Generation Computer Systems* 26, 6 (2010), 847–856.

[15] William Gropp and Ewing Lusk. 2004. Fault Tolerance in Message Passing Interface Programs. *Int. J. High Perform. Comput. Appl.* 18, 3 (Aug. 2004), 363–372.

[16] J. Ignacio Hidalgo, Juan Lanchares, Francisco Fernández de Vega, and Daniel Lombraña. 2007. Is the Island Model Fault Tolerant?. In *Proceedings of the 9th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '07).* ACM, New York, NY, USA, 2737–2744.

[17] J. L. J. Laredo, A. E. Eiben, M. van Steen, P. A. Castillo, A. M. Mora, and J. J. Merelo. 2008. P2P Evolutionary Algorithms: A Suitable Approach for Tackling Large Instances in Hard Optimization Problems. In *Euro-Par 2008 – Parallel Processing*, Emilio Luque, Tomàs Margalef, and Domingo Benítez (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 622–631.

[18] J. L. J. Laredo, A. E. Eiben, M. van Steen, and J. J. Merelo. 2010. EvAg: a scalable peer-to-peer evolutionary algorithm. *Genetic Programming and Evolvable Machines* 11, 2 (01 Jun 2010), 227–246.

[19] Wei-Po Lee. 2007. Parallelizing evolutionary computation: A mobile agent-based approach. *Expert Systems with Applications* 32, 2 (2007), 318–328.

[20] T. T. Magalhães, Nascimento L. H. C., E. K. Silva, Augusto D. A., and H. J. C. Barbosa. 2014. Hybrid metaheuristics for optimization using a parallel islands model. XXXV CILAMCE – Ibero-Latin American Congress on Computational Methods in Engineering.

[21] T. T. Magalhães, E. K. Silva, and H. J. C. Barbosa. 2015. Migration policies to improve exploration in parallel island models for optimization via metaheuristics. XXXVI CILAMCE – Ibero-Latin American Congress on Computational Methods in Engineering.

[22] Andrea Mambrini and Dirk Sudholt. 2014. Design and Analysis of Adaptive Migration Intervals in Parallel Evolutionary Algorithms. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO '14).* ACM, New York, NY, USA, 1047–1054.

[23] Guenter Obiltschnig. 2004–2017. POrtable COmponents (POCO) C++ Libraries. https://pocoproject.org/. (2004–2017).

[24] Michael O'Neill and Conor Ryan. 2003. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language.* Kluwer Academic Publishers, Norwell, MA, USA.

[25] B. Paechter, T. Back, M. Schoenauer, M. Sebag, A. E. Eiben, J. J. Merelo, and T. C. Fogarty. 2000. A Distributed Resource Evolutionary Algorithm Machine (DREAM). In *Proceedings of the 2000 Congress on Evolutionary Computation.*

[26] D. Robilliard, V. Marion-Poty, and C. Fonlupt. 2009. Genetic programming on graphics processing units. *Genetic Programming and Evolvable Machines* 10, 4 (2009), 447.

[27] Kenneth Sörensen, Marc Sevaux, and Fred Glover. 2017. A History of Metaheuristics. *CoRR* abs/1704.00853 (2017).

[28] W. Richard Stevens. 1990. *UNIX Network Programming.* Prentice Hall.

[29] E.-G. Talbi. 2009. *Metaheuristics: From Design to Implementation.* Wiley Publishing.

[30] E-G. Talbi, J-M. Geib, Z. Hafidi, and D. Kebbal. 1998. *A fault-tolerant parallel heuristic for assignment problems.* Springer Berlin Heidelberg, Berlin, Heidelberg, 306–314.

[31] Douglas Thain, Todd Tannenbaum, and Miron Livny. 2005. Distributed Computing in Practice: The Condor Experience: Research Articles. *Concurr. Comput. : Pract. Exper.* 17, 2-4 (Feb. 2005), 323–356.

[32] Marco Tomassini. 2005. *Spatially Structured Evolutionary Algorithms: Artificial Evolution in Space and Time (Natural Computing Series).* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[33] J. M. Whitacre, R. A. Sarker, and Q. T. Pham. 2008. The Self-Organization of Interaction Networks for Nature-Inspired Optimization. *Trans. Evol. Comp* 12, 2 (April 2008), 220–230.

[34] W. R. M. U. K. Wickramasinghe, Maarten van Steen, and A. E. Eiben. 2007. Peer-to-peer evolutionary algorithms with adaptive autonomous selection. In *GECCO.* ACM, 1460–1467.

[35] D. S. Wilks. 2006. *Statistical methods in the atmospheric sciences.* 2nd Ed. International Geophysics Series, Vol. 59. Academic Press. 627 p.