Advances in Engineering Software 119 (2018) 116–133

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

JACK2: An MPI-based communication library with non-blocking synchronization for asynchronous iterations

Frédéric Magoulès, Guillaume Gbikpi-Benissan[*]

CentraleSupélec, Université Paris-Saclay, 3 rue Joliot-Curie, Gif-sur-Yvette 91190, France

A R T I C L E I N F O

Keywords:
Distributed computing
Message passing interface
Iterative methods
Asynchronous iterations
Convergence detection

A B S T R A C T

In this paper, we address the design of a communication library which particularly targets distributed iterative computing, including randomly executed asynchronous iterations. The well-known MPI programming framework is considered, upon which unique generic routines are proposed for both blocking and non-blocking communication modes. This allows for developing unique software applications to experiment both classical and asynchronous iterative methods through the same programming pattern. Convergence detection issues are investigated on pure algorithmic aspects, from which we provide an efficient exact approach to compute global convergence residual norms, by means of non-blocking synchronization. Point-to-point message reception and sending requests are carefully handled in view of producing the least possible delays about transmitted iterative data. Extensive experiments with an existing MPI-based scientific application validate both the proposed MPI-like programming framework and design options to achieve best performances of asynchronous iterative computing.

1. Introduction

Asynchronous iterations have been formally introduced by Chazan and Miranker [1] as a computational model which allows a full overlapping of communication and computation phases during parallel iterations. Such numerical methods thus feature two main interests for high performance computing (HPC), especially in dynamic heterogeneous environments. First, relatively to the scalability of distributed applications, the standard upper bound on the expectable acceleration factor (see Amdahl [2]) does not apply, thanks to the fact that there is no strictly serial portions, which is reversely the case for classical iterations where communication ensures sequential consistency. This moreover implies non-determinism and flexible data dependency between parallel processes, hence raises a second interest related to their low sensitivity to unbalanced workload and temporary resource failures. Asynchronous iterations are thus gaining much attention nowadays (see, e.g., [3–6]), as they are inherently natural candidates for massively parallel computing.

Nevertheless, despite a concise corresponding algorithmic pattern, implementation of asynchronous iterations requires much more than a straightforward update of classical iterations loops, depending however on the communication middleware under use. Moreover, because of their non-deterministic behavior, the accurate evaluation of a residual-based stopping criterion becomes an issue, as introduced by Bertsekas and Tsitsiklis [7]. One thus has to handle one, possibly more, convergence detection protocol(s), according to both effectiveness and efficiency expectation (see, e.g., [8–12]). Few libraries therefore emerged to help easily assess asynchronous iterative methods. Jace [13] is a message passing environment based on the Java Remote Method Invocation (RMI) middleware. It provides communication routines which naturally fit asynchronous iterations semantics. The library is continuously improved, mainly in view of resource failure support (see [14]), and it now includes the convergence detection algorithm from Charr et al. [15], which is an extension of Bahi et al. [12] to volatile computing environments (see [16]). As an alternative to Java performance limits, Crac [17] is a similar C++ communication library mainly targeting grid computing environments. There as well, a proper message passing behavior is developed to take full advantage of asynchronism and multi-site cluster architectures. Although the Message Passing Interface (MPI) standard is undoubtedly the most popular communication framework for parallel scientific computing, it has been designed on general basis for all kinds of distributed algorithms. Jace and Crac therefore do not rely on MPI, since the peculiar communication pattern of asynchronous iterations is not easy to handle in such a framework, even less in a non-intrusive manner which would totally encapsulate the management of pure communication-related objects. Recently though, as far as we know, Jack [18] has been proposed as the first successful attempt to build a C++ MPI-based communication

[*] Corresponding author.
E-mail addresses: frederic.magoules@hotmail.com (F. Magoulès), guillaume.benissan@centralesupelec.fr (G. Gbikpi-Benissan).

https://doi.org/10.1016/j.advengsoft.2018.01.009
Received 31 August 2017; Received in revised form 14 November 2017; Accepted 22 January 2018
Available online 13 February 2018

library for both classical and asynchronous scientific iterative computing. It provides an application programming interface (API) very close to MPI routines, including the definition of a new type of communication request devoted to the automatic reception of any incoming message during computation phases. The last received data are however delivered into the computation reception buffer only when explicitly requested. Not surprisingly, some communication objects still have to be managed outside the library, just as in classical MPI-based programming.

The main bottom line is that all of these libraries handle the asynchronous convergence detection issue through various heuristics based on local convergence of each process, therefore they do not guarantee effective convergence state after termination. We present here Jack2, an extended version of Jack, which provides a completely encapsulating new API, and allows exact convergence testing through actual global residual norm computation. The current paper is based upon Gbikpi-Benissan and Magoulès [19], but it additionally includes: (i) further details and discussion about the termination protocol, (ii) flexibility and usability improvements in the architecture of the library, (iii) complete implementation details about components, (iv) and more experimental results.

In Section 2, we describe the algorithmic framework induced by the class of distributed iterative methods under consideration. Computational models are recalled, then asynchronous convergence detection algorithms are discussed. Section 3 focuses on the message-passing framework and induced programming patterns. We first illustrate classical MPI-based applications, then propose various Jack2-based counterparts. Section 4 gives further details about the architecture of the library and some key implementation options. Behavior of components is described, regarding both point-to-point and collective communication. In Section 5 at last, we analyze corresponding experimental results obtained in an homogeneous computational environment. The case of linear system resolution is considered, where we measure efficiency impacts of various configurations, as well as convergence accuracy.

2. Theoretical background

2.1. Distributed iterative computing

Let us consider a vector space E and some given sequence of mappings

$$f^{(k)}: E \to E, \quad k \in \mathbb{N},$$

where in many cases, we might have $f^{(0)} = f^{(1)} = f^{(2)} = \cdots$. For instance, consider a linear problem

$$Ax = b,$$

and a Jacobi splitting of the matrix A, which yields iterations:

$$Dx^{k+1} = (D - A)x^k + b,$$

where D is the diagonal part of A. Then, here, one would have:

$$f^{(0)}(x) = f^{(1)}(x) = \cdots = f^{(k)}(x) = D^{-1}[(D - A)x + b].$$

Let each $f^{(k)}$ be decomposed on a set of p processes, $p \in \mathbb{N}$, $p > 0$, such that

$$f^{(k)}(x) = \begin{bmatrix} f_1^{(k)}(x_1, \ldots, x_p) & \cdots & f_p^{(k)}(x_1, \ldots, x_p) \end{bmatrix}^{\mathsf{T}}, \quad x = [x_1 \quad \cdots \quad x_p]^{\mathsf{T}},$$

with

$$f_i^{(k)}: E \to E_i, \quad i \in \{1, \ldots, p\}, \qquad E = E_1 \times \cdots \times E_p.$$

Now, let $\{P^{(k)}\}_{k\in\mathbb{N}}$ be a sequence of sets, with

$$P^{(k)} \subseteq \{1, \ldots, p\}, \quad \forall\, k \in \mathbb{N},$$

and let as well $\tau_{i,\,j}$, with $i, j \in \{1, \ldots, p\}$, be nonnegative integer-valued functions such that

$$\tau_{i,j}(k) \le k, \quad \forall\, k \in \mathbb{N}.$$

We address in this paper the design and implementation of a communication library for general (asynchronous) distributed iterations generating a sequence $\{x^k\}_{k\in\mathbb{N}}$ of vectors such that

$$x_i^{k+1} = \begin{cases} f_i^{(k)}\left(x_1^{\tau_{i,1}(k)}, \ldots, x_p^{\tau_{i,p}(k)}\right), & i \in P^{(k)}, \\ x_i^k, & i \in \{1, \ldots, p\} \setminus P^{(k)}. \end{cases} \tag{1}$$

The sequence $\{x^k\}_{k\in\mathbb{N}}$ is expected to converge toward a set of solution vectors $S^* \subset E$, i.e.,

$$\exists\, k_0 \in \mathbb{N}: \qquad \forall\, k \ge k_0, \quad x^k \in S^*.$$

Each function $\tau_{i,j}$ defines the successive versions of the jth component which have been used to update the ith component. If we have, for instance, $\tau_{3,1}(10) = 6$, this means that at the 10th iteration, the 3rd component (held by the 3rd process) is updated by using the version of the 1st component computed (by the 1st process) at the 6th iteration. This thus defines a delay of $10 - 6 = 4$ iterations on the 1st component. We note that the successive sets $P^{(k)}$ implies that each time at least one component is updated, the global, abstract, iteration counter k is incremented. Fig. 1 illustrates such an asynchronous scheme. We refer to Chau [20] for an exhaustive survey about conditions which ensure the convergence of the sequence. Mostly, theoretical results have been established for contracting fixed-point mappings, under the natural assumptions that no process definitively stops either computing or receiving new computation data.

The computational model of classical iterative methods is given by specifically defining

$$P^{(k)} = \{1, \ldots, p\}, \quad \tau_{i,j}(k) = k, \qquad \forall\, k \in \mathbb{N},$$

which simplifies the model (1) into

$$x_i^{k+1} = f_i^{(k)}(x_1^k, \ldots, x_p^k), \quad i \in \{1, \ldots, p\}. \tag{2}$$

Algorithm 1 describes a trivial parallel procedure implementing the computational model (2). For each iteration on each process, computation is performed first, then messages are sent and received. In the MPI framework, any process willing to send data to a slower one thus needs to wait until the latter finishes its computation phase and requests the reception of the message. Such dedicated communication phases could therefore considerably decrease the parallel computational efficiency. A well known improvement consists in Algorithm 2 which overlaps communication and computation phases. In this scheme, message reception is requested from the beginning of the iteration, therefore a process can start sending data immediately after its computation phase even when the destination process is still computing. This requires a little more memory space, since the buffers x_j^{k+1} and x_j^k are simultaneously accessed, but the overall computation time is generally decreased compared to the trivial scheme. Indeed, there is almost no more time dedicated to communication on slowest processes, which, on the other hand, does not provide much performance gain when the workload is perfectly balanced.

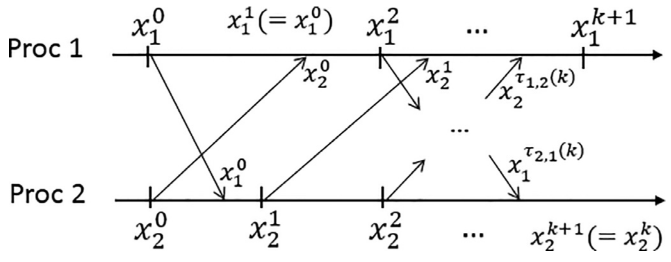

Fig. 1. Asynchronous iterative computation.

```
1:  $k := 0$
2:  repeat
3:      $x_i^{k+1} := f_i^{(k)}\left(x_1^k, \ldots, x_p^k\right)$
4:      for all $j \in \{1, \ldots, i-1, i+1, \ldots, p\}$ do
5:          Request reception of $x_j^{k+1}$ from process $j$
6:          Request sending of $x_i^{k+1}$ to process $j$
7:      end for
8:      Wait for communication completion
9:      $k := k + 1$
10: until $x^k \in S^*$
11: return $x^k$
```

Algorithm 1. Trivial parallel iterative scheme.

```
1:  $k := 0$
2:  repeat
3:      for all $j \in \{1, \ldots, i-1, i+1, \ldots, p\}$ do
4:          Request reception of $x_j^{k+1}$ from process $j$
5:      end for
6:      $x_i^{k+1} := f_i^{(k)}\left(x_1^k, \ldots, x_p^k\right)$
7:      for all $j \in \{1, \ldots, i-1, i+1, \ldots, p\}$ do
8:          Request sending of $x_i^{k+1}$ to process $j$
9:      end for
10:     Wait for communication completion
11:     $k := k + 1$
12: until $x^k \in S^*$
13: return $x^k$
```

Algorithm 2. Overlapping parallel iterative scheme.

```
1:  $k_i := 0$
2:  **repeat**
3:      **for all** $j \in \{1, \dots, i-1, i+1, \dots, p\}$ **do**
4:          Request reception of $x_j^{\tau_{i,j}(k)}$ from process $j$
5:      **end for**
6:      $x_i^{k_i+1} := f_i^{(k)}\left(x_1^{\tau_{i,1}(k)}, \dots, x_i^{k_i}, \dots, x_p^{\tau_{i,p}(k)}\right)$
7:      **for all** $j \in \{1, \dots, i-1, i+1, \dots, p\}$ **do**
8:          Request sending of $x_i^{k_i+1}$ to process $j$
9:      **end for**
10:     $k_i := k_i + 1$
11: **until** $\left(x_1^{k_1}, \dots, x_p^{k_p}\right) \in S^*$
12: **return** $\left(x_1^{k_1}, \dots, x_p^{k_p}\right)$
```

Algorithm 3. Asynchronous parallel iterative scheme.

Yet, to reach a full overlapping algorithmic scheme where processes never idle while waiting for communication completion, both the sets $P^{(k)}$ and the functions $\tau_{i,\,j}$ are left randomly defined by every execution of Algorithm 3.

Quite simply, as soon as a process terminates an iteration, it starts a next one without waiting for the completion of communication requests. If new data were not received, latest ones can just be used in the next computation phase. Note that the global iteration variable k here remains abstract.

2.2. Asynchronous convergence detection

The loop stopping criterion in Algorithm 3 (line 11) ensures that the returned vector surely belongs to the set S^* of admissible solutions. It requires however to explicitly build a global vector $\bar{x} = (x_1^{k_1}, \dots, x_p^{k_p})$ which, then, will be evaluated as a potential solution. Therefore, most of the convergence detection protocols available in this research field only approximate this stopping criterion. The supervised termination approach from Savari and Bertsekas [10] is currently the sole decentralized way of handling a global vector during distributed asynchronous iterations. It relies on distributed snapshot algorithms introduced by Chandy and Lamport [21], but avoids the condition of first-in-first-out (FIFO) message delivering without use of piggybacking or acknowledgement techniques. We refer to Kshemkalyani et al. [22] for an introductory overview about distributed snapshot.

The decentralized supervised termination protocol from Savari and Bertsekas [10] consists of a coordination phase designed upon tree network topology, followed by a partial snapshot phase, which is directly performed on the initial communication graph. A spanning tree may be obtained from this graph at the initialization of the application. Suppose now that each process is able to evaluate some local criterion, as, for instance,

$$x_i^{k_i+1} \simeq x_i^{k_i},$$

which will be considered as indicating local convergence. In the coordination phase, local convergence is notified from the leaves to the root of the tree, as described by Algorithm 4. When local convergence happens on a leaf process (having no children in the communication tree), a notification is sent to its father in the tree. Internal processes (having children and father) do the same if, additionally, all of their children have notified local convergence. Under same two conditions, the root process instead triggers the partial snapshot phase by recording its current local component $\bar{x}_i = x_i^k$ and sending it to its one-hop neighbor processes in the initial communication graph. Rules given by Algorithm 5 are then followed to isolate a global potential solution $\bar{x}$ distributed over all of the processes. Here as well, local convergence must be once more verified by non-root processes to make them record their local component, but additionally, only if they had already received the recorded component of at least one of their neighbor processes. This necessarily happens, as the root process sends its recorded component to initiate the partial snapshot phase. Each local component is recorded only once and immediately sent to neighbor processes. Any process i thus records a component $\bar{x}_i$ and receives recorded components $\bar{x}_j$, with $j \neq i$, from all of its neighbors. It is finally a classical, application-dependent, matter to check if this distributed global vector $\bar{x}$ is an admissible solution.

Few remarks directly follow. First, the coordination phase is not required to be able to build $\bar{x}$, even though it is useful to avoid unnecessarily performing snapshots at a stage where convergence is not yet likely to be reached. Similarly, a snapshot could be arbitrarily initiated, regardless local convergence conditions, just as a collective operation requested by the application. This leads to Algorithm 6, which thus introduces a general partial snapshot operation where each process records some local data and sends it (or a part of it) to its neighbor processes.

```
1:  if $x_i^{k_i+1} \simeq x_i^{k_i}$ then
2:    if leaf process then
3:      Send notification to father
4:      Enter snapshot phase
5:    end if
6:    if internal process then
7:      if notification received from all children then
8:        Send notification to father
9:        Enter snapshot phase
10:     end if
11:   end if
12:   if root process then
13:     if notification received from all children then
14:       $\bar{x}_i := x_i^{k_i+1}$
15:       for all neighbors $j \neq i$ in the communication graph do
16:         Send $\bar{x}_i$ to $j$
17:         Enter snapshot phase
18:       end for
19:     end if
20:   end if
21: end if
```

Algorithm 4. Coordination phase of supervised termination.

```
1: if $x_i^{k_i+1} \simeq x_i^{k_i}$ then
2:     if $\bar{x}_j$ received from a neighbor $j \neq i$ and $\bar{x}_i$ undefined then
3:         $\bar{x}_i := x_i^{k_i+1}$
4:         for all neighbors $j \neq i$ do
5:             Send $\bar{x}_i$ to $j$
6:         end for
7:     end if
8: end if
9: if $\bar{x}_j$ received from all neighbors $j \neq i$ and $\bar{x}_i$ defined then
10:     return $\bar{x}$
11: end if
```

Algorithm 5. Partial snapshot phase of supervised termination.

```
1: if $\bar{x}_i$ undefined then
2:     $\bar{x}_i := x_i^{k_i+1}$
3:     for all neighbors $j \neq i$ do
4:         Send $\bar{x}_i$ to $j$
5:     end for
6: end if
7: if $\bar{x}_j$ received from all neighbors $j \neq i$ and $\bar{x}_i$ defined then
8:     return $\bar{x}$
9: end if
```

Algorithm 6. General partial snapshot operation.

```
while (res_vec_norm >= 1e-8) {
  for (i = 0; i < numb_neighb; i++) {
    MPI_Irecv(recv_buf2[i], rbuf_size[i], dtype, neighb_rank[i],
        tag, comm, &(recv_request[i]));
  }
  Copy(sol_vec_buf, sol_vec_buf2);
  Compute(recv_buf, sol_vec_buf);
  Map(sol_vec_buf, send_buf);
  for (i = 0; i < numb_neighb; i++) {
    MPI_Isend(send_buf[i], sbuf_size[i], dtype, neighb_rank[i],
        tag, comm, &(send_request[i]));
  }
  for (i = 0; i < sol_vec_size; i++) {
    res_vec_buf[i] = abs(sol_vec_buf[i] - sol_vec_buf2[i]);
  }
  MPI_Allreduce(MPI_IN_PLACE, res_vec_buf, res_vec_size, dtype,
      MPI_MAX, comm);
  res_vec_norm = max(res_vec_buf);
  MPI_Waitall(numb_neighb, recv_request, MPI_STATUSES_IGNORE);
  MPI_Waitall(numb_neighb, send_request, MPI_STATUSES_IGNORE);
  Swap(recv_buf, recv_buf2);
}
```

Listing 1. MPI-based overlapping scheme.

Actually, such an operation is really meaningful as a non-blocking global synchronization of parallel processes, which allows one to completely overlap computation and global synchronization phases, hence to run random asynchronous iterations with same stopping criteria used in the classical synchronous context. It is also important to note that, while in this theoretical description, each process i sends its local solution $\bar{x}_i$, in practical situations, it actually only needs to send interface data consistent with $\bar{x}_i$.

3. Implementation of distributed iterative methods

3.1. MPI programming framework

Let us consider, for instance, a straightforward MPI-based implementation of Algorithm 2 (line 2–12), as shown in Listing 1. Both declaration and initialization of variables are omitted, as they can be easily deduced. A procedure *Compute* encloses the main computation part, which corresponds to

$$x_i^{k+1} := f_i^{(k)}\left(x_1^k, \dots, x_p^k\right).$$

The variable *sol_vec_buf* (solution vector buffer) would therefore contain x_i^k data as input, and then x_i^{k+1} data as output, while the variable *recv_buf* would contain the set $\{x_j^k\}_{j \neq i}$. Using the procedure *Copy*, x_i^k is priorly saved into the temporary variable *sol_vec_buf2*, in view of a comparison between x_i^k and x_i^{k+1}. Next, a procedure *Map* is called to set a message sending buffer for each neighbor process, based on the newly computed solution. Here, for example, all of the sending buffers would contain the same x_i^{k+1} data. As stopping criterion, it is usual for many iterative methods to verify that

$$x^{k+1} \in S^* \quad \Leftrightarrow \quad \|x^{k+1} - x^k\| \simeq 0,$$

where $\|.\|$ denotes a given norm. In this example, we consider the maximum norm given by:

$$\|x\|_\infty = \max_{1 \le i \le n} |x_i|, \quad x \in \mathbb{C}^n.$$

A function *abs* is used to get the absolute value (or the module) of a real (or a complex) number, while a function *max* gives the maximum entry of an array. The keyword *MPI_IN_PLACE* indicates that the variable *res_vec_buf* (residual vector buffer) is used as both input and output of the collective reduction operation. Finally, the procedure *Swap* exchanges two pointers, which is useful to avoid copy of data when handling the temporary buffers *recv_buf2*.

Classically, MPI routines *MPI_Irecv* and *MPI_Isend* are used to request message reception and sending, respectively, while the norm-based stopping criterion is evaluated by means of the collective procedure *MPI_Allreduce*. As suggested however in Chau et al. [23], it would be more efficient to take advantage of persistent requests *MPI_Recv_init* and *MPI_Send_init*. This reduces the routines invocation overhead, as shown by Listing 2, even if, on the other hand, it prevents one from swapping pointers on reception buffers.

Implementing asynchronous iterative methods within the MPI framework obviously raises nontrivial questions related to the management of successive communication requests, the management of associated buffers and the evaluation of such a stopping criterion. On performance side, for instance, it might be preferable to allow for several message reception requests during computation phase, in order to use the least delayed data. On the other hand, one notices that overlapping (Algorithm 2) and asynchronous schemes (Algorithm 3) nearly follow the same algorithmic pattern. It could therefore be desirable to maintain a unique implementation code where a parameter will be checked in order to produce, at runtime, either classical or random asynchronous iterations.

```
for (i = 0; i < numb_neighb; i++) {
  MPI_Recv_init(recv_buf2[i], rbuf_size[i], dtype, neighb_rank[i],
      tag, comm, &(recv_request[i]));
  MPI_Send_init(send_buf[i], sbuf_size[i], dtype, neighb_rank[i],
      tag, comm, &(send_request[i]));
}
while (res_vec_norm >= 1e-8) {
  MPI_Startall(numb_neighb, recv_request);
  Copy(sol_vec_buf, sol_vec_buf2);
  Compute(recv_buf, sol_vec_buf);
  Map(sol_vec_buf, send_buf);
  MPI_Startall(numb_neighb, send_request);
  for (i = 0; i < sol_vec_size; i++) {
    res_vec_buf[i] = abs(sol_vec_buf[i] - sol_vec_buf2[i]);
  }
  MPI_Allreduce(MPI_IN_PLACE, res_vec_buf, res_vec_size, dtype,
      MPI_MAX, comm);
  res_vec_norm = max(res_vec_buf);
  MPI_Waitall(numb_neighb, recv_request, MPI_STATUSES_IGNORE);
  MPI_Waitall(numb_neighb, send_request, MPI_STATUSES_IGNORE);
  Copy(recv_buf2, recv_buf);
}
```

Listing 2. MPI-based overlapping scheme with persistent requests.

3.2. *Jack2 programming framework*

In a study from Balaji et al. [24] about the scalability of MPI-based applications, the authors pointed out the fact that the MPI specification does not allow one to initially provide information about communication graphs, while this could help optimizing communication resources. Such a graph should however be handled in a distributed way to avoid memory overhead costs. Somehow though, Chau et al. [23] expressed a similar programming pattern through the use of persistent requests, where message buffers and neighbor processes are specified once for all before the iterations loop (see Listing 2). Jack2 follows same ideas to provide a complete initialization part for repetitive communication parameters. Of course, our design remains flexible enough to allow several ways of interacting with the library, which as well implies several levels of encapsulation of communication objects.

Full initialization of a Jack2 communicator object therefore requires to explicitly specify variables about communication graph, communication buffers and residual norm evaluation. Listing 3 describes an implementation of the overlapping scheme within the Jack2 framework.

As soon as the communicator object is initialized with communication graph and buffers, incoming communication channels are opened via MPI non-blocking reception requests, so that each process is ready to receive computation data from any of its neighbors. The method *Recv* of the class *JACKSyncComm* is meant to deliver pending messages received from all of the neighbors, then to throw new reception requests. The method *Finalize* therefore ensures the cancellation of the last, unfulfilled, ones. Relatively to the MPI-based implementation, the temporary reception buffers *recv_buf2* are now internally managed by the library, and collective reduction is made persistent too. The application thus does not need to handle any pure communication-related object, barring communicators themselves. One notices, as minor drawback, the method *Send* which is blocking when invoked on a Jack2 synchronous communicator. This prevents one from overlapping message sending and residual norm computation. Our send/receive semantic however follows the put/get paradigm advocated by related works, e.g., Bahi et al. [13] (Jace), Couturier and Domas [17] (Crac).

With such an API, it becomes quite straightforward to handle communication requests during random asynchronous iterations. Considering a communicator of type *JACKAsyncComm*, only lines 11–15 of Listing 3 would be rearranged, as shown in Listing 4 (lines 6–12). The initialization part of the communicator is omitted, since the same interface is shared by both *JACKAsyncComm* and *JACKSyncComm*. Here, the methods *Recv* and *Send* are non-blocking, and the library regulates underlying MPI requests itself. Of course, messages can be received many times between two successive *Recv* calls. The non-blocking version of the method *AllReduce* returns a flag indicating whether the reduction operation is completed or not. Its input-output parameter *res_vec_buf* can be safely accessed once the flag is armed, then the next *AllReduce* call automatically starts a new collective reduction. Nonetheless, in order to ensure correct termination, one needs to add the partial snapshot operation previously described by Algorithm 6. Lines 1, 2 and 8–24 of Listing 5 indicate a possible pattern related to the integration of such a non-blocking collective operation. Just like the method *AllReduce*, an armed flag is returned to indicate the end of the snapshot operation. When so, the variables *ss_sol_vec_buf* and *ss_recv_buf* respectively contain $\bar{x}_i$ and $\{\bar{x}_j\}_{j\neq i}$ data, so that invoking the procedure *Compute* allows one to perform a global iteration

$$y_i :=f_i^{(k)}(\bar{x}_1,\dots,\bar{x}_p), \qquad \forall\ i \in \{1,\dots,p\},$$

whereupon each local residual vector is set to $|y_i - \bar{x}_i|$. A flag-type variable *reduce_flag* is then armed to make the process enter the reduction phase which will evaluate the global residual norm as

```
JACKSyncComm<double,int> sync_comm;
sync_comm.Init(mpi_comm);
sync_comm.InitRecv(recv_buf, rbuf_size, neighb_rank, numb_neighb);
sync_comm.InitSend(send_buf, sbuf_size, neighb_rank, numb_neighb);
sync_comm.InitAllReduce(res_vec_buf, res_vec_buf, res_vec_size,
    MPI_MAX);
while (res_vec_norm >= 1e-8) {
  Copy(sol_vec_buf, sol_vec_buf2);
  Compute(recv_buf, sol_vec_buf);
  Map(sol_vec_buf, send_buf);
  sync_comm.Send();
  for (i = 0; i < sol_vec_size; i++) {
    res_vec_buf[i] = abs(sol_vec_buf[i] - sol_vec_buf2[i]);
  }
  sync_comm.AllReduce();
  res_vec_norm = max(res_vec_buf);
  sync_comm.Recv();
}
sync_comm.Finalize();
```

Listing 3. Jack2-based overlapping scheme.

$\max_{1\le i\le p} \max |y_i - \bar{x}_i|$.

It thus turns out that the stopping criterion implies:

$$(x_1^{k_1},...,x_p^{k_p}) \in S^* \quad \Leftrightarrow \quad \|f^{(k_0)}(x_1^{k_1},...,x_p^{k_p}) - (x_1^{k_1},...,x_p^{k_p})\|_\infty \simeq 0,$$

with $k_0 \le k$, and $k_i \le k$, $\forall\ i \in \{1, ..., p\}$.

Finally, to produce a unique code for both overlapping and asynchronous schemes, the top front-end interface of Jack2 provides two classes, *JACKComm* and *JACKConv*, which manage communication requests and convergence detection, respectively. On message sending and reception aspects, *JACKComm* obviously features the same init/send/recv interface shared by *JACKSyncComm* and *JACKAsyncComm*. Methods *SwitchSync* and *SwitchAsync* allow one to choose either blocking or non-blocking mode at any time. On the other hand, a general convergence detection object needs one more layer of abstraction, due to the differences introduced for handling the termination of asynchronous iterations. Moreover, we previously saw that initiating snapshots at an early stage of the iterative process unnecessarily holds communication resources. The convergence detector therefore implements the whole supervised termination protocol given by Algorithms 4 and 5. This leads to one another collective routine which encompasses the coordination, snapshot and reduction phases, and thus additionally requires an input flag-type parameter indicating at any time whether the process is under local convergence or not. We mention however that the synchronous mode directly jumps to the reduction phase. Still, Listing 5 makes two calls of the procedure *Compute*, the second one being for performing an iteration with the snapshot-based solution vector. Being so intrusive makes it hard to design a standalone

```
while (res_vec_norm >= 1e-8) {
  Copy(sol_vec_buf, sol_vec_buf2);
  Compute(recv_buf, sol_vec_buf);
  Map(sol_vec_buf, send_buf);
  async_comm.Send();
  async_comm.AllReduce(&end_flag);
  if (end_flag) {
    res_vec_norm = max(res_vec_buf);
    for (i = 0; i < sol_vec_size; i++) {
      res_vec_buf[i] = abs(sol_vec_buf[i] - sol_vec_buf2[i]);
    }
  }
  async_comm.Recv();
}
```

Listing 4. Jack2-based asynchronous scheme without accurate termination.

```
async_comm.InitSnapshot(ss_sol_vec_buf, ss_recv_buf, sol_vec_buf,
    sol_vec_size);
reduce_flag = 0;
while (res_vec_norm >= 1e-8) {
  Copy(sol_vec_buf, sol_vec_buf2);
  Compute(recv_buf, sol_vec_buf);
  Map(sol_vec_buf, send_buf);
  async_comm.Send();
  if (!reduce_flag) {
    async_comm.Snapshot(&end_flag);
    if (end_flag) {
      Copy(ss_sol_vec_buf, sol_vec_buf2);
      Compute(ss_recv_buf, ss_sol_vec_buf);
      for (i = 0; i < sol_vec_size; i++) {
        res_vec_buf[i] = abs(ss_sol_vec_buf[i] - sol_vec_buf2[i]);
      }
      reduce_flag = 1;
    }
  } else {
    async_comm.AllReduce(&end_flag);
    if (end_flag) {
      res_vec_norm = max(res_vec_buf);
      reduce_flag = 0;
    }
  }
  async_comm.Recv();
}
```

Listing 5. Jack2-based asynchronous scheme.

method which would enclose the whole global residual norm evaluation. Listing 6 instead describes a programming pattern which shows the possibility to keep the application code separated from the convergence detection procedure.

Here now, once a snapshot is completed, buffers are temporarily swapped (lines 20 and 21) such that, at the next iteration, variables *sol_vec_buf* and *recv_buf* actually contain the output of the snapshot, and hence the residual vector buffer is rightly filled up using globally consistent data. Once done, the variables are swapped back (lines 12 and 13), in order to avoid disrupting the main iterative process. Snapshot buffers and swapping procedures could thus be managed by the standalone residual norm evaluation routine if we explicitly distinguish local and global residual data. More precisely, the library can internally make a copy of the residual vector buffer at this iteration where it contains snapshot-based data, and then use its own internal buffer as both input and output of the reduction operation. Listing 7 describes the resulting general Jack2-based programming pattern for both classical and random asynchronous iterations.

4. Architecture and design of the communication library

4.1. Overall features

As a C++ library, Jack2 follows, as much as possible, object-oriented paradigms thoroughly integrated by the Unified Modeling Language (UML) [25], which is one of the top popular theoretical tools for software modeling. Fig. 2 proposes an implementation-level UML class diagram describing the architecture of the library. Even though the classes *JACKSyncComm* and *JACKAsyncComm* share the same types of operations, switching between blocking (*SwitchSync*) and non-blocking (*SwitchAsync*) communication routines is a specific behavior of *JACKComm*, which therefore is not a simple interface or an abstract class. Instead, delegation is considered by means of function pointers which allows us to invoke methods of the appropriate class instance, according to the communication mode, without using *if* conditions. The same pattern applies to classes *JACKConv, JACKSyncConv* and *JACKAsyncConv*. Both communicators and convergence detectors rely on instances of a class *JACKAllReduce*. For blocking modes, this simply results in *MPI_Allreduce* calls, which will be the case for non-blocking modes too, relatively to the *MPI_Iallreduce* routine, in versions supporting the third MPI specification. Our own non-blocking reduction implementation is based on a distributed, non-deterministic, leader election protocol designed over acyclic graphs (see, e.g., [26, Section 4.4.3], or [18]). A class *JACKSpanningTree* therefore builds a distributed spanning tree upon the application communication graph. If no graph information is specified, an all-to-all connectivity (complete graph) is assumed, as currently implied by the MPI standard. The communication tree is produced by means of an echo algorithm [27], which, contrarily to a structural minimum spanning tree, effectively minimizes traversals time by taking into account actual communication

```
swap_flag = 0;
reduce_flag = 0;
while (res_vec_norm >= 1e-8) {
  Copy(sol_vec_buf, sol_vec_buf2);
  Compute(recv_buf, sol_vec_buf);
  Map(sol_vec_buf, send_buf);
  async_comm.Send();
  if (swap_flag) {
    for (i = 0; i < sol_vec_size; i++) {
      res_vec_buf[i] = abs(sol_vec_buf[i] - sol_vec_buf2[i]);
    }
    Swap(recv_buf, ss_recv_buf);
    Swap(sol_vec_buf, ss_sol_vec_buf);
    reduce_flag = 1;
    swap_flag = 0;
  }
  if (!reduce_flag) {
    async_comm.Snapshot(&end_flag);
    if (end_flag) {
      Swap(recv_buf, ss_recv_buf);
      Swap(sol_vec_buf, ss_sol_vec_buf);
      swap_flag = 1;
    }
  } else {
    async_comm.AllReduce(&end_flag);
    if (end_flag) {
      res_vec_norm = max(res_vec_buf);
      reduce_flag = 0;
    }
  }
  async_comm.Recv();
}
```

Listing 6. Non-intrusive Jack2-based asynchronous scheme.

delays. At last, classes *JACKSnapshot* and *JACKSnapshotSB96* respectively implement the general partial snapshot given by Algorithm 6 and the snapshot-based supervised termination from Savari and Bertsekas [10], described by both Algorithms 4 and 5.

Jack2 does not launch any thread for its non-blocking collective operations. Instead, the execution of the underlying distributed protocols progresses each time a call is made to the corresponding routine. Considering the non-blocking MPI operations which are almost always running in background, this prevents the library for introducing additional concurrency against the computation part of the application, which therefore can remain the main executed steps during asynchronous iterations. Users thus have enough flexibility to regulate themselves the execution frequency of Jack2 objects. Next sections give further details about their specific behavior.

4.2. Point-to-point communication

While it is commonly admitted that, during random execution of asynchronous iterations, processes should benefit from the least delayed data, the classical corresponding put/get paradigm implies to only fill up the application buffers with the last of the messages received between two successive calls of the message reception routine. To be more precise, let us consider again the computational model (1), and take, for instance,

$$E = \mathbb{C}^n, \qquad E_i = \mathbb{C}^{n_i}, \quad i \in \{1,\dots,p\}, \qquad \sum_{i=1}^{p} n_i = n.$$

Actually, convergence could be guaranteed for a more general model featuring:

$$x_l^{k+1} = f_l^{(k)}\left(x_1^{\rho_{l,1}(k)},\dots,x_n^{\rho_{l,n}(k)}\right), \quad l \in N^{(k)}, \tag{3}$$

with, as well,

$$\rho_{l,t}(k) \le k, \quad \forall\ l,\ t \in \{1,\dots,n\}, \qquad N^{(k)} \subseteq \{1,\dots,n\}.$$

Nonetheless, delivering only last received messages consists of particularly setting:

```
JACKComm<double,int> comm;
JACKConv<double,int> conv;
comm.Init(mpi_comm);
comm.InitRecv(recv_buf, rbuf_size, neighb_rank, numb_neighb);
comm.InitSend(send_buf, sbuf_size, neighb_rank, numb_neighb);
conv.Init(mpi_comm, neighb_rank, numb_neighb, neighb_rank,
    numb_neighb);
conv.InitAllReduce(res_vec_buf, res_vec_buf, res_vec_size,
    MPI_MAX);
if (async_flag) {
  conv.InitSnapshot(&sol_vec_buf, sol_vec_size, &recv_buf,
      rbuf_size, send_buf, sbuf_size, &local_conv_flag);
  comm.SwitchAsync();
  conv.SwitchAsync();
}
while (res_vec_norm >= 1e-8) {
  Copy(sol_vec_buf, sol_vec_buf2);
  Compute(recv_buf, sol_vec_buf);
  Map(sol_vec_buf, send_buf);
  comm.Send();
  for (i = 0; i < sol_vec_size; i++) {
    res_vec_buf[i] = abs(sol_vec_buf[i] - sol_vec_buf2[i]);
  }
  local_conv_flag = (max(res_vec_buf) < 1e-8);
  conv.SnapReduce(&end_flag);
  if (end_flag) {
    res_vec_norm = max(res_vec_buf);
  }
  comm.Recv();
}
```

Listing 7. Jack2-based overlapping/asynchronous scheme.

$$\rho_{l,1}(k)=\cdots=\rho_{l,n_1}(k),\quad \ldots,\quad \rho_{l,n-n_p+1}(k)=\cdots=\rho_{l,n}(k),\quad \forall\, l\in\{1,\ldots,n\},$$
$$\rho_{1,t}(k)=\cdots=\rho_{n_1,t}(k),\quad \ldots,\quad \rho_{n-n_p+1,t}(k)=\cdots=\rho_{n,t}(k),\quad \forall\, t\in\{1,\ldots,n\}.$$

To be able to effectively implement the model (3) while allowing the possibility to have $\rho_{l_2,t}(k) > \rho_{l_1,t}(k)$, with $l_1 < l_2$ and both l_1 and l_2 belonging to, for instance, the set $\{1, \ldots, n_1\}$, Jack2 instead delivers any incoming message directly into the corresponding message reception buffer of the application. Therefore, at a given loop iteration, processes can possibly use more recent data for last updated local components, instead of only benefiting from data received at the beginning of the computation phase. We mention that, with asynchronous iterations, there is no more consistency concerns when accessing reception buffers, therefore, completion test is no more mandatory for providing a correct implementation. Still, this implies at least that the MPI library does not fill the reception buffer with random data each time a communication is requested, which would seem to be, more generally, a useless memory operation. In any case, however, such an approach for message reception provides more flexibility since here also, users still have the possibility to manage two reception buffers if they rather wish (or need) to avoid including data on the fly.

To achieve that, Jack2 is parameterized to activate several MPI reception requests on each reception buffer provided by the user application. The MPI specification guarantees FIFO delivering on communication links which are identified by the triplet communicator/source/tag. Therefore, the method *Recv* of the class *JACKAsyncComm* is only meant to reactivate completed requests, as shown for instance by Listing 8.

Users are allowed to configure such a parameter (the number of reception requests per neighbor) which defines a maximum message reception rate. This implies on the other hand that allowing a higher message sending rate could lead to a counter performance, as the number of pending MPI sending requests may quickly increase, which would yield much more delayed iterations data. Jack2 therefore discards message sending requests on busy communication links, as one can see in Listing 9.

With a sufficiently high message reception rate, all of the message sending requests are theoretically able to be satisfied. We however note that, on homogeneous computing resources anyway, it is unlikely to have a process running several iterations while its neighbor is completing only one. On such platforms, very few incoming messages are therefore expectable during computation phases. In any case, discarding message sending requests can be unavoidable if data

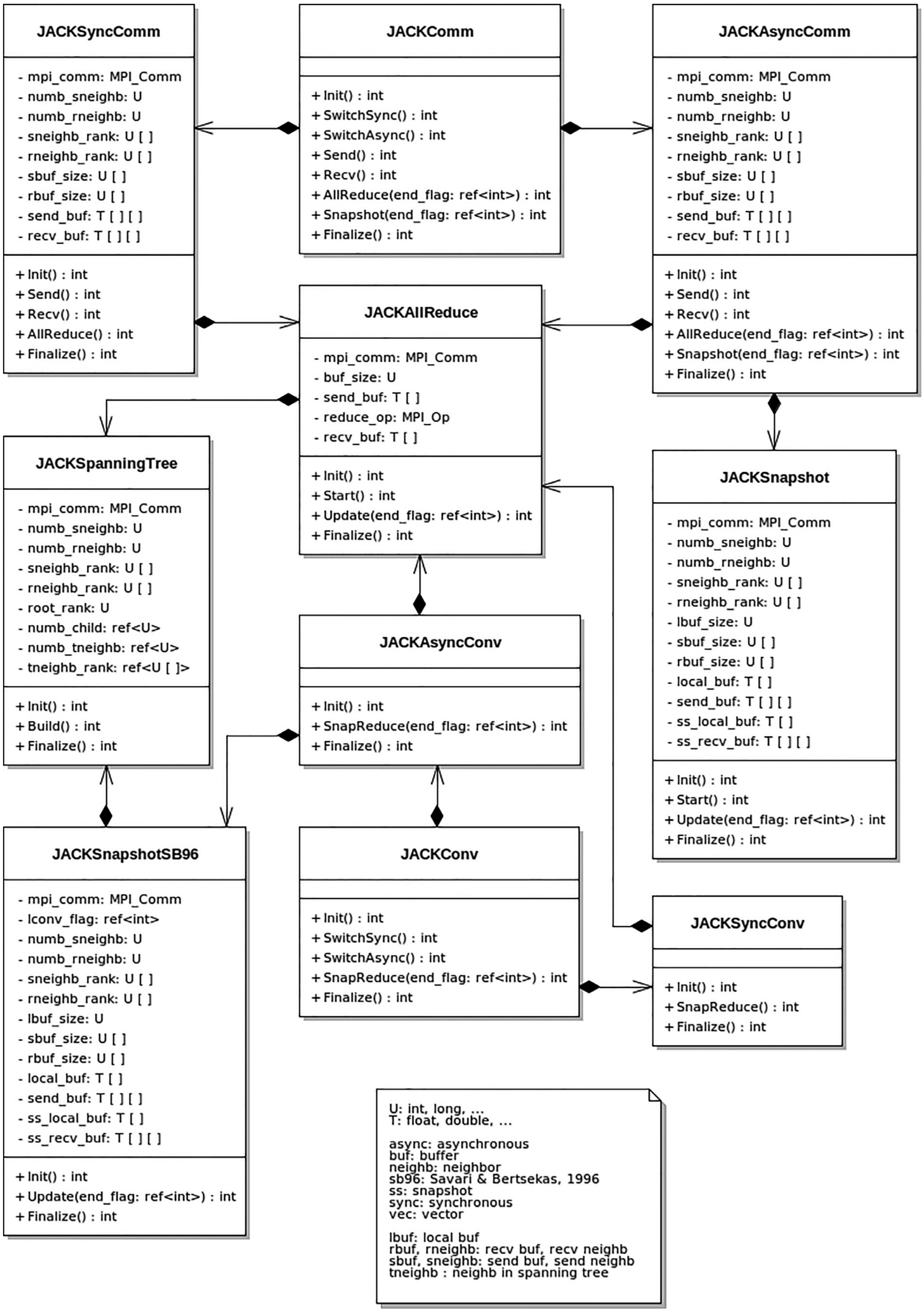

Fig. 2. Main classes featuring the basic interface of Jack2.

```
for (i = 0; i < numb_rneighb; i++) {
  for (j = 0; j < numb_req_per_rneighb; j++) {
    MPI_Test(&(recv_req[i][j]), &flag, MPI_STATUS_IGNORE);
    if (flag) {
      MPI_Start(&(recv_req[i][j]));
    }
  }
}
```

Listing 8. Method *Recv* of class *JACKAsyncComm*.

transmission itself actually covers at least two loop iterations, which somehow exhibits advantages of asynchronous iterations. In the put/get paradigm, while a message is buffered for being sent, it can be replaced by a new *put* call. This can be achieved here through the MPI synchronous mode (*MPI_Issend*), since data are not buffered by the MPI library, if it follows the standard recommendation, and hence, the actual message sending buffer is directly updated by the user application (procedure *Map* in Listing 7, for instance).

4.3. Collective operations

We shall finally focus on non-blocking collective operations, which are not executed within parallel threads, but instead progress at the pace of explicit requests from the user application. These classes are designed from a state machine view, such that each time their method *Update* is called, a set of apt conditions (seen as occurring events) is checked in order to step forward in the execution of the distributed protocol. Each particular set of concurrently expected events therefore defines a state of the running class instance. Here as well, in order to avoid *if* conditions, a function pointer is moved throughout a set of private methods implementing each behavioral state of the class.

Fig. 3 presents an UML statechart diagram of the class *JACKAllReduce*. Just like in the MPI specification, this operation successively consists of a reduction and a broadcast phases. We recall that the reduction operation is performed along with a tree-based leader election protocol described in Lynch [26, Section 4.4.3] (also see, e.g., [18]). The broadcast messages then follow the inverse path of the reduction messages. Upon start request, processes enter an *UpdateReduce* state where three types of event are expected. Leaves in the tree potentially trigger the transition toward an *UpdateBCast* state, since they have only one neighbor in the tree, and therefore at least satisfy the condition "*numb_reduce_recv* = *numb_tneighb* - *1*". They thus send their local input data to that neighbor. The other processes loop on the *UpdateReduce* state while combining their local data with received ones, according to the type of reduction, until they evolve toward either the *UpdateBCast* or the initial *Idle* state. We should recall that the looping transition and the transition to *UpdateBCast* are based on concurrent events, so that the former might be triggered while the *when* condition of the latter is satisfied, which would results in reaching the other condition "*numb_reduce_recv* = *numb_tneighb*". Actually, this may happen to only one of the processes (the unique elected "leader"), which thus initiates the broadcast phase. While being in the *UpdateBCast* state, processes thus expect a broadcast message (containing the result of the reduction operation) from the neighbor to which they previously sent their reduction message. The received data are then forwarded to their other neighbors in the tree, and so on to the leaves. It may however happen that all of the processes enter this *UpdateBCast* state. In this case, two of them (the elected "co-leaders") receive a last reduction message from each other, upon which they both deduce the result of the reduction and broadcast it to their other neighbors in the tree, then also return to the *Idle* state.

The protocols implemented as classes *JACKSnapshot* and *JACKSnapshotSB96* have already been fully described through Algorithm 4–Algorithm 6. As summarized by Fig. 4, the general partial snapshot exhibits only one set of expected events, which results in two quite simple states. It however clearly points out the fact that the message sending buffer provided by the user application must always contain consistent data, regarding the solution vector buffer (which is the local buffer here). This implies, for instance in Listing 6, that the procedure *Map* should always precede the call of the method *Snapshot*. The same remark also applies to the more elaborated snapshot-based supervised termination, which is implemented by means of the statechart diagram in Fig. 5. There, from an *UpdateCoordStart* state, the leaves of the spanning tree instantly evolve toward an *UpdateSShotStart* state, if they are under local convergence. Otherwise, they wait in an *UpdateCoordEnd* state for local convergence. Except the root of the tree, the other processes trigger the same transitions once they have received a coordination message from all of their children. The root process instead initiates the snapshot phase and enters an *UpdateSShotEnd* state. Once processes in the *UdpateSShotStart* state observe again local convergence, they evolve to the *UpdateSShotEnd* state too, but only if they received at least one snapshot message which triggered the looping transition. Finally, from *UpdateSshotEnd*, processes switch back to *UpdateCoordStart* once all of their expected snapshot messages have been received.

```
for (i = 0; i < numb_sneighb; i++) {
  MPI_Test(&(send_req[i]), &flag, MPI_STATUS_IGNORE);
  if (flag) {
    MPI_Start(&(send_req[i]));
  }
}
```

Listing 9. Method *Send* of class *JACKAsyncComm*.

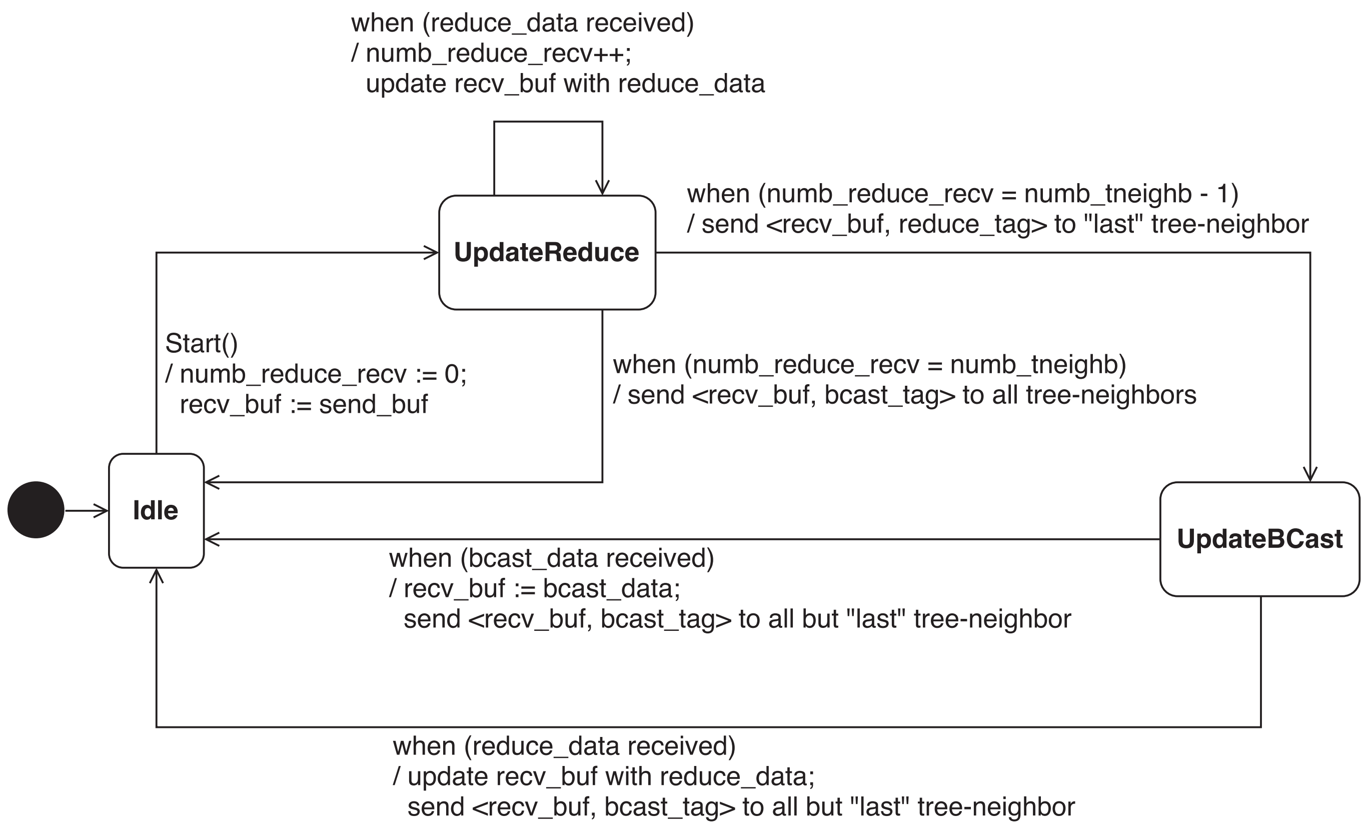
when (reduce_data received)
/ numb_reduce_recv++;
update recv_buf with reduce_data
UpdateReduce
when (numb_reduce_recv = numb_tneighb - 1)
/ send <recv_buf, reduce_tag> to "last" tree-neighbor
Start()
/ numb_reduce_recv := 0;
recv_buf := send_buf
when (numb_reduce_recv = numb_tneighb)
/ send <recv_buf, bcast_tag> to all tree-neighbors
Idle
UpdateBCast
when (bcast_data received)
/ recv_buf := bcast_data;
send <recv_buf, bcast_tag> to all but "last" tree-neighbor
when (reduce_data received)
/ update recv_buf with reduce_data;
send <recv_buf, bcast_tag> to all but "last" tree-neighbor

Fig. 3. States and behavior of a *JACKAllReduce* instance.

5. Numerical experiments

5.1. Problem and experimental settings

To validate the proposed framework, we plugged Jack2 into an existing scientific application devoted to the parallel resolution of convection-diffusion problems of the form

$$\frac{\partial u}{\partial t} - \nu \Delta u + a.\ \nabla u = s,$$

where u is the unknown function defined on $\mathbb{R}^+ \times ([0, 1])^3$. The host software application implements regular finite-differences and backward Euler discretization schemes, which results in solving successive sparse linear systems over small constant time steps. The three-dimensional domain $([0, 1])^3$ is decomposed in a two-dimensional coarse grid on the (x,y)-plane, while parallel solutions are computed by

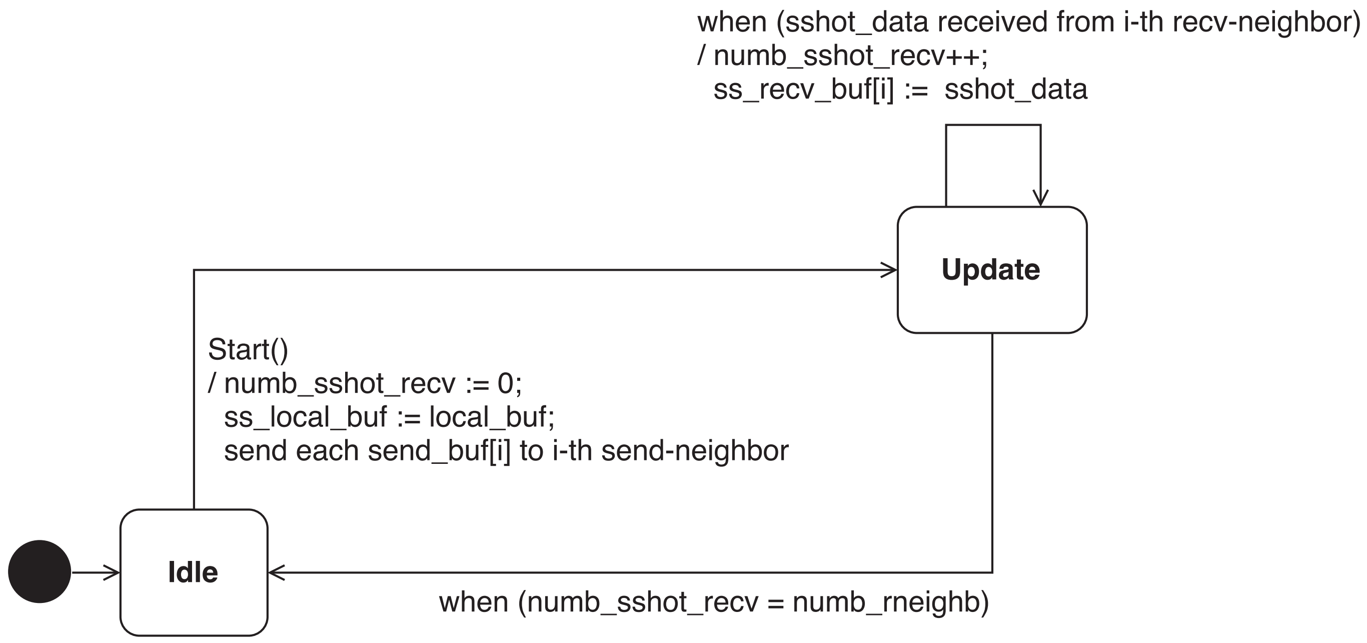
when (sshot_data received from i-th recv-neighbor)
/ numb_sshot_recv++;
ss_recv_buf[i] := sshot_data
Update
Start()
/ numb_sshot_recv := 0;
ss_local_buf := local_buf;
send each send_buf[i] to i-th send-neighbor
Idle
when (numb_sshot_recv = numb_rneighb)

Fig. 4. States and behavior of a *JACKSnapshot* instance.

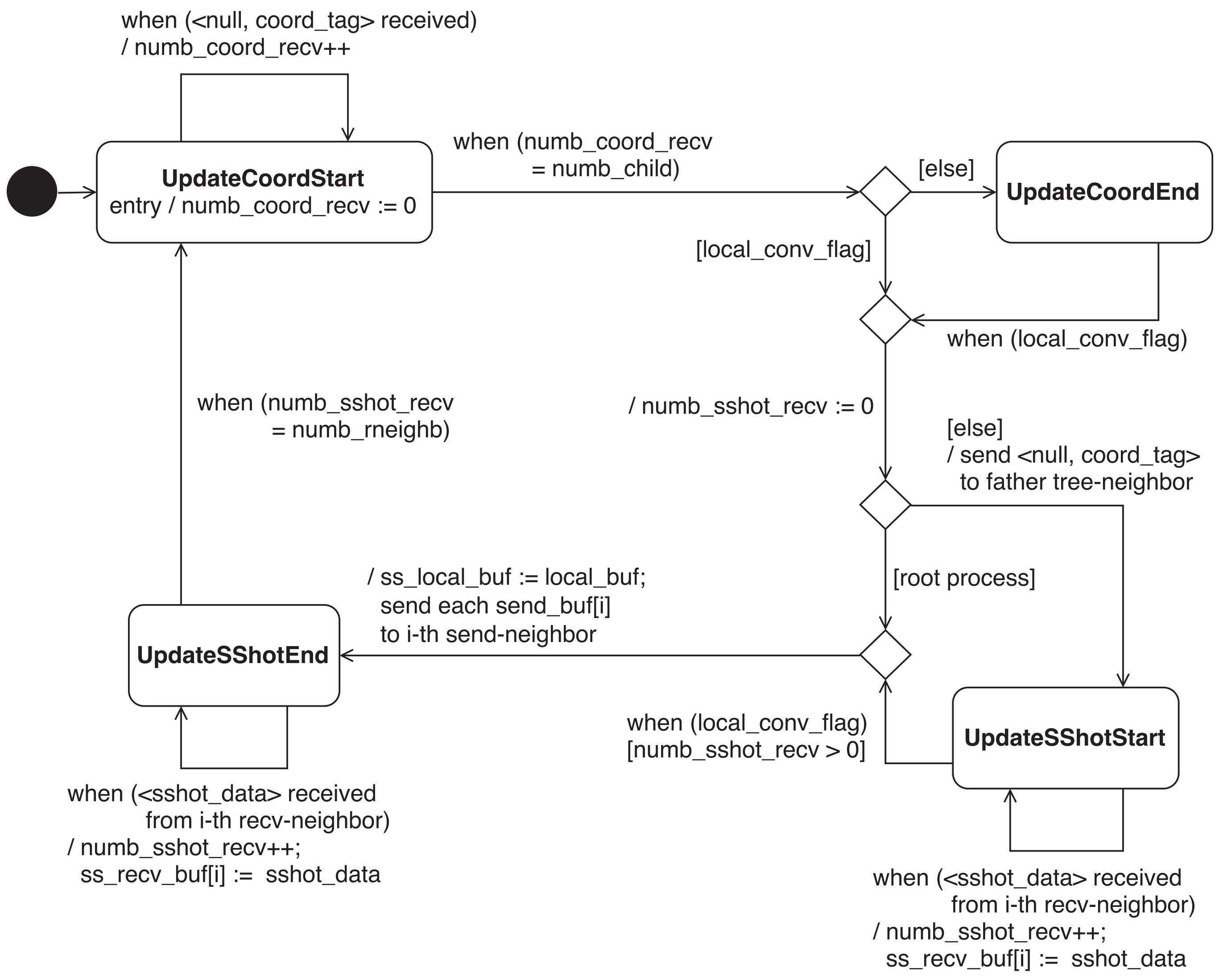
when (<null, coord_tag> received)
/ numb_coord_recv++
UpdateCoordStart
entry / numb_coord_recv := 0
when (numb_coord_recv = numb_child)
[else]
UpdateCoordEnd
[local_conv_flag]
when (local_conv_flag)
when (numb_sshot_recv = numb_rneighb)
/ numb_sshot_recv := 0
[else] / send <null, coord_tag> to father tree-neighbor
[root process]
/ ss_local_buf := local_buf; send each send_buf[i] to i-th send-neighbor
UpdateSShotEnd
when (local_conv_flag) [numb_sshot_recv > 0]
UpdateSShotStart
when (<sshot_data> received from i-th recv-neighbor) / numb_sshot_recv++; ss_recv_buf[i] := sshot_data
when (<sshot_data> received from i-th recv-neighbor) / numb_sshot_recv++; ss_recv_buf[i] := sshot_data

Fig. 5. States and behavior of a *JACKSnapshotSB96* instance.

Table 1
Comparison between *MPI_Issend*-based and *MPI_Isend*-based implementations of Jack2, with $p = 192$, $n = 180^3$.

	MPI_Issend				MPI_Isend			
rpn	$r \times 10^7$	wt	k	unsent	$r \times 10^7$	wt	k	unsent
1	6.21	239	146,440	84,579	–	> 4 h	–	–
2	6.61	236	144,723	747	6.38	228	140,157	356
3	6.67	236	144,243	533	5.90	228	140,327	341
4	6.28	236	144,325	543	6.31	228	139,779	349
5	6.46	236	144,392	532	6.03	229	140,035	362
20	6.61	237	144,256	626	5.91	229	140,624	344
50	6.25	241	144,958	586	6.63	232	141,332	350

following a non-overlapping additive Schwarz pattern [28,29] where Gauss–Seidel iterations are performed inside each subdomain. The main program sets up the corresponding band-diagonal matrices, each band being stored as a separate one dimensional array, then sequentially runs a time iterations loop wherein the linear system solver is called by each of the parallel processes. Communication requests inside the solver procedure have been reshaped from the MPI to the Jack2 framework. Each process handles exactly one subdomain, and the number of processes always equals the number of processor cores used.

Experiments have been led on an SGI ICE X supercomputer with FDR Infiniband network (56 Gbit/s). Each node consists of two Intel Haswell Xeon CPUs with 12 cores (24 cores per node) at 2.30 GHz, and 60 GB RAM allocated to running jobs, and the MPI library SGI-MPT has been loaded as communication middleware. Subsequent results summarize a wide set of experimental measurements related to each resolution of a sparse linear system of the form

$$Ax = b, \quad x \in \mathbb{R}^n,$$

arising from the discretization of the convection-diffusion equation. Results feature:

- residual norms $r: = \|A\tilde{x} - b\|_\infty$, where $\tilde{x}$ is the returned solution,
- execution times in seconds, denoted as “wt” (wall-clock time),
- numbers of iterations $k := \max_{1 \le i \le p}\{k_i\}$, where k_i is the number of iterations on the ith process (we see from Table 1 that, for asynchronous iterations, the maximum number did quite well correlate with the execution time),
- numbers of snapshots, denoted as “ss”,
- and maximum numbers of discarded send requests, denoted as “unsent”.

Experimental parameters include the problem size n, the number of MPI processes p, the maximum admissible residual norm $r^* = 10^{-6}$, and the maximum number of active MPI reception requests concurrently

Table 2
Reference results from synchronous iterations, for $n = 185^3$.

p	$r \times 10^7$	wt	k
144	8.33	812	280,953
192	8.32	626	282,702
240	8.31	510	284,118

Table 3
Comparison between convergence detection procedures, with $n = 185^3$.

	SnapReduce (SB96 [10])				Snapshot + AllReduce (SSAR)			
p	$r \times 10^7$	wt	k	ss	$r \times 10^7$	wt	k	ss
144	6.45	746	320,940	154	9.05	620	345,531	14,382
192	6.24	575	340,176	135	6.56	476	365,986	14,282
240	6.03	463	343,527	147	8.07	377	366,854	10,096

handled for each neighbor of a process, denoted as “rpn” (requests per neighbor).

5.2. Numerical results

We first focus on items discussed about point-to-point communication management. Table 1 reports some results illustrating the general trend observed, while corresponding synchronous iterations featured $r = 8.33\times 10^{-7}$, wt = 255 and k = 128,504. With only one active reception request per neighbor, the *MPI_Isend*-based implementation of Jack2 made the solver procedure exceed the limit wall-clock time set to 4 h. While looking at the corresponding results for the *MPI_Issend*-based implementation, we see that the number of discarded message sending requests was tremendously high, revealing that up to 48% of iterative updates remained unsent. As expected, with a non-buffering routine though (*MPI_Issend*), actual sending buffers were continuously updated even if requests were dismissed, which yielded less delays on dependencies. It followed that the execution time did not depend at all on the MPI sending request rate. Note however that, since the computational platform is homogeneous, processes performed their iterations approximately at the same pace, which required no more than two concurrently active reception requests per neighbor for making the *MPI_Isend*-based implementation fully effective.

A second set of experiments particularly addresses convergence detection matters and the efficiency of the corresponding collective routines. We consider here, on one hand, the method *SnapReduce* from Listing 7, which implements a protocol from Savari and Bertsekas [10] (Algorithms 4 and 5), and on the other hand, the programming pattern from Listing 6 which makes direct use of the general methods *Snapshot* (Algorithm 6) and *AllReduce* (based on leader election [26, Section 4.4.3]). Table 2 first reports reference results related to the synchronous solver, then Table 3 highlights comparative performance of the two asynchronous scheme programming patterns. While the coordination phase of the SB96 protocol led to very few numbers of snapshots, this, however, was not relevant in our experimental configuration for performing faster. Instead, beside induced termination delays, we can even notice that SB96-based iterations ran longer than those relying on the SSAR approach, since results feature lower numbers of iterations despite higher execution times. This came from the fact that, in the SSAR approach, local residual norm is not required to be computed at every iteration. It therefore turned out that the communication overhead costs introduced by regular and frequent snapshot executions were negligible compared to the impact of the systematic evaluation of a local convergence criterion. One advantage of the SB96 approach, though, lies in its quite stable final residual norm which dwelt here strictly between 6×10^{-7} and 7×10^{-7}, for a threshold set to 10^{-6}.

6. Conclusion

Taking full advantage of asynchronous iterations is still a challenging computational matter which possibly requires more suitable parallel programming patterns. By designing specialized libraries, researchers therefore aim to provide efficient asynchronous iterations environments requiring lowest possible implementation costs for users. In this paper, we addressed the development of Jack2, an MPI-based communication library for distributed iterative computing. Key results are twofold. First, regarding the convergence detection issue arising from random execution of asynchronous iterations, existing distributed approaches involve some local convergence criterion, as researchers rightly argued so far that it is ineffective to check global convergence at early stages of the computation where local convergence is not even somehow persistent on any of the processes. This perfectly matches the classical parallel programming pattern where a local convergence residual norm is evaluated at each iteration. By considering here an exact general snapshot-based approach to directly evaluate consistent global residual norms, we showed that systematic irrelevant local residual norm computation could be, on the contrary, far more time consuming than a distributed protocol uselessly executed concurrently with the main computation phase. Such an important practical aspect seems to have been missed till now.

Beside convergence detection matters, we focused on minimizing delays on transmitted data too. From obtained experimental execution times, it came out that best performances could be reached by suitably combining outgoing message buffering with the MPI synchronous communication mode. This is currently under consideration for further experiments.

On implementation costs aspects, Jack2 features an API quite close to the MPI specification. As it was the case for validating the library, this makes it easy to upgrade existing MPI-based applications toward the Jack2 programming framework, wherein a handful set of tools is already available for the experimental study of asynchronous iterations.

Acknowledgement

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

[1] Chazan D, Miranker W. Chaotic relaxation. Linear Algebra Appl 1969;2(2):199–222.
[2] Amdahl GM. Validity of the single processor approach to achieving large scale computing capabilities. Proceedings of the Spring Joint Computer Conference, April 18–20, 1967, Atlantic City, NJ, USA. AFIPS ’67 (Spring). 1967. p. 483–5.
[3] Chau M, Couturier R, Bahi J, Spiteri P. Parallel solution of the obstacle problem in grid environments. Int J High Perform Comput Appl 2011;25(4):488–95.
[4] Zhang Y, Gao Q, Gao L, Wang C. Maiter: an asynchronous graph processing framework for delta-based accumulative iterative computation. IEEE Trans Parallel Distrib Syst 2014;25(8):2091–100.
[5] Wolfson-Pou J, Chow E. Reducing communication in distributed asynchronous iterative methods. Procedia Comput Sci 2016;80:1906–16.
[6] Magoulès F, Venet C. Asynchronous iterative sub-structuring methods. Math Comput Simul 2018;145(Supplement C):34–49.
[7] Bertsekas DP, Tsitsiklis JN. Convergence rate and termination of asynchronous iterative algorithms. Proceedings of the 3rd International Conference on Supercomputing, 5–9 June 1989, Crete, Greece. New York, NY, USA: ACM; 1989. p. 461–70.
[8] Bertsekas DP, Tsitsiklis JN. Some aspects of parallel and distributed iterative algorithms – a survey. Automatica 1991;27(1):3–21.
[9] El Baz D. A method of terminating asynchronous iterative algorithms on message passing systems. Parallel Algorithms Appl 1996;9(1–2):153–8.
[10] Savari SA, Bertsekas DP. Finite termination of asynchronous iterative algorithms. Parallel Comput 1996;22(1):39–56.
[11] Evans DJ, Chikohora S. Convergence testing on a distributed network of processors. Int J Comput Math 1998;70(2):357–78.
[12] Bahi JM, Contassot-Vivier S, Couturier R. An efficient and robust decentralized algorithm for detecting the global convergence in asynchronous iterative algorithms. High Performance Computing for Computational Science - VECPAR 2008. Lecture Notes in Computer Science. Vol. 5336. Springer Berlin Heidelberg; 2008. p. 240–54.
[13] Bahi J, Domas S, Mazouzi K. Jace: a java environment for distributed asynchronous

iterative computations. 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing. 11–13 February 2004, A Coruña, Spain. 2004. p. 350–7.

[14] Bahi JM, Couturier R, Vuillemin P. Jacev: a programming and execution environment for asynchronous iterative computations on volatile nodes. High Performance Computing for Computational Science - VECPAR 2006. Lecture Notes in Computer Science. Vol. 4395. Springer Berlin Heidelberg; 2007. p. 79–92.

[15] Charr J-C, Couturier R, Laiymani D. A decentralized and fault tolerant convergence detection algorithm for asynchronous iterative algorithms. J Supercomput 2010;53(2):269–92.

[16] Charr J-C, Couturier R, Laiymani D. JACEP2P-V2: A fully decentralized and fault tolerant environment for executing parallel iterative asynchronous applications on volatile distributed architectures. Advances in Grid and Pervasive Computing: 4th International Conference, GPC 2009, Geneva, Switzerland, May 4–8, 2009. Proceedings. Springer Berlin Heidelberg; 2009. p. 446–58.

[17] Couturier R, Domas S. Crac: a grid environment to solve scientific applications with asynchronous iterative algorithms. IEEE International Parallel and Distributed Processing Symposium. 26–30 March 2007, Long Beach, California, USA. 2007. p. 1–8.

[18] Magoulès F, Gbikpi-Benissan G. JACK: An asynchronous communication kernel library for iterative algorithms. J Supercomput 2017;73(8):3468–87.

[19] Gbikpi-Benissan G, Magoulès F. JACK2: A New High-Level Communication Library for Parallel Iterative Methods. In: Iványi P, Topping B, Várady G, editors. Proceedings of the Fifth International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering. Stirlingshire, UK: Civil-Comp Press; 2017. . doi:10.4203/ccp.111.39.

[20] Chau M. Algorithmes ParallÈles Asynchrones Pour la Simulation Numérique. Toulouse, France: Institut National Polytechnique de Toulouse; 2005. Ph.D. thesis.

[21] Chandy KM, Lamport L. Distributed snapshots: determining global states of distributed systems. ACM Trans Comput Syst 1985;3(1):63–75.

[22] Kshemkalyani AD, Raynal M, Singhal M. An introduction to snapshot algorithms in distributed computing. Distrib Syst Eng 1995;2(4):224–33.

[23] Chau M, El Baz D, Guivarch R, Spiteri P. MPI implementation of parallel subdomain methods for linear and nonlinear convectionâ;;diffusion problems. J Parallel Distrib Comput 2007;67(5):581–91.

[24] Balaji P, Buntinas D, Goodell D, Gropp W, Kumar S, Lusk E, Thakur R, Träff JL. MPI on a million processors. In: Ropo M, Westerholm J, Dongarra J, editors. Recent Advances in Parallel Virtual Machine and Message Passing Interface: 16th European PVM/MPI Users' Group Meeting, Espoo, Finland, September 7–10, 2009. Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg; 2009. p. 20–30.

[25] Rumbaugh J, Jacobson I, Booch G. The unified modeling language reference manual. Object technology series 1. Boston, USA: Addison-Wesley; 1999.

[26] Lynch NA. Distributed Algorithms. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc; 1996.

[27] Chang EJH. Echo algorithms: depth parallel operations on general graphs. IEEE Trans Software Eng 1982;SE-8(4):391–401.

[28] Widlund O, Dryja M. An Additive Variant of the Schwarz Alternating Method for the Case of Many Subregions. Tech Rep. New York, USA: Department of Computer Science, Courant Institute; 1987. Technical Report 339, Ultracomputer Note 131.

[29] Gander MJ. Schwarz methods over the course of time. Electron Trans Numer Anal 2008;31:228–55.