J Supercomput (2017) 73:3468–3487
DOI 10.1007/s11227-016-1702-2

JACK: an asynchronous communication kernel library for iterative algorithms

Frédéric Magoulès[1] · Guillaume Gbikpi-Benissan[2]

Published online: 22 March 2016

Abstract This article presents a new communication library developed to ease the implementation of both asynchronous and synchronous iterative methods. A mathematical and algorithmic framework about fixed-point methods is described to introduce this class of parallel iterative algorithms, although this library can be used for a larger class of parallel algorithms. After an overview of the main features, we describe detailed implementation aspects arising from the asynchronous context. While the library is mainly based on top of Message Passing Interface library, it has been designed to be easily extended to other types of communication middleware. Finally, some numerical experiments validate this new library, used for implementing both a classical parallel scheme and a sub-structuring approach of the Jacobi iterative method.

Keywords Asynchronous method · Iterative method · Sub-structuring method · Parallel computing · Distributed computing

1 Introduction

With the evolution of hardware and networks, it is more and more easy to link various computational units together and create a powerful computing machine. Such systems can even be decentralized on geographically distant sites. Since the ability to reuse old hardware or to extend existing cluster with newer machines is of great interest, those

✉ Guillaume Gbikpi-Benissan
guillaume.benissan@irt-systemx.fr

Frédéric Magoulès
frederic.magoules@hotmail.com

[1] CentraleSupélec, Université Paris-Saclay, Grande Voie des Vignes, Paris, France

[2] Technological Research Institute SystemX, 8 Avenue de la Vauve, 91120 Palaiseau, France

clusters are usually composed of heterogeneous machines linked together through an heterogeneous communication network. However, these systems bring up new problems such as the efficient management of the heterogeneous communication links and the use of low-speed interconnections.

Iterative parallel computing requires a lot of global synchronizations between processes. These synchronizations are very expensive due to the high latencies of the networks. They also imply a lot of wasted time if the workloads are not well balanced. The heterogeneous nature of the machines makes such load balancing very hard to achieve, resulting in higher time loss, compared to homogeneous machines. Hence, being able to resort to asynchronous iterations seems interesting to overcome that communication limitation. The principle of this scheme is that every computational unit can perform iterations of the algorithm without waiting for incoming data messages. While there is much to gain from asynchronous iterations, they require more conditions on the algorithms to ensure convergence. The research on these algorithms is quite notable in parallel computing, especially for distributed and grid computing. Performance comparisons between synchronous and asynchronous iterative methods on parallel computers have been investigated since 1978 [6,23]. Many studies confirmed the efficiency of asynchronous methods for solving various mathematical problems, such as the obstacle problem [14,36], optimization, and flow problems [12,21]. Large linear systems have been handled too, especially through multi-splitting methods [2,11,18].

Nevertheless, programming and experimenting an asynchronous iterative method may lead to manage a lot of implementation options, relative to both the computation environment and the communication middleware. It could thus be highly profitable to use an appropriate communication library that makes it easy to produce both synchronous and asynchronous implementations of an iterative method. Some libraries have been recently developed, like Java Asynchronous Computing Environment (JACE) for Java applications [1,5], or Communication Routines for Asynchronous Computations (CRAC) for C++ as well [3,17]. However, these libraries use standard Transmission Control Protocol (TCP) connections, and are optimized for use in a grid environment only, which means that they are more difficult to use when the interconnection is not made through an Ethernet network. Even on the Ethernet, they do not take advantage of Remote Direct Memory Access (RDMA-capable) interfaces. Classical supercomputers use high-speed interconnections like InfiniBand or Fibre Channel, which are not directly compatible with the TCP. While there exists some ways of emulating the TCP on these interconnections, such an adaptation decreases the performance, especially by increasing the latency. Although all these low-level compatibility issues are handled by the well-known Message Passing Interface (MPI) libraries, the MPI specifications are not natively designed for easily programming asynchronous iterations.

In this paper, we propose a new C/C++ MPI-based library, namely JACK (Just an Asynchronous Communication Kernel), for quick and effective developments of both synchronous and asynchronous iterative algorithms. In Sect. 2, we present a general mathematical and algorithmic context that can be targeted by the JACK Application Programming Interface (API). While not being a requirement for the library usability, we consider here, as an example, the resolution of algebraic linear systems by means of fixed-point methods. Next, Sect. 3 introduces main features of the JACK library.

An overview of the API shows how iterative methods can be implemented for parallel computing, and particularly when asynchronous iterations are to be experimented. The problem of terminating asynchronous iterations is briefly exposed, along with the state-of-art solution provided by JACK. Details are then given about the handling of communication requests, and about some basics of the convergence detection protocol. Finally, in Sect. 4, we report experimental results on parallel executions of the Jacobi algorithm, for a comparative study of the use of asynchronous iterations with a sub-structuring method. We briefly recall the Jacobi algorithm and its different parallel implementations that is considered. Main experimental features are then described before to discuss execution time measures issues.

2 Mathematical and algorithmic framework

First asynchronous iterative algorithms were derived from fixed-point methods based on contracting operators including Jacobi, Richardson, Successive Over Relaxation method, fixed step gradient descent [7,24]. Other asynchronous algorithms include multi-splitting methods [9,10,30]. We thus present the asynchronous computing model through its native mathematical framework, although the efficient use of such a library is more tightly related to the algorithmic modeling.

2.1 Iterative methods

Let n be an integer in $\mathbf{N}^*$, let A be a given nonsingular $n \times n$ matrix in $\mathbf{R}^{n\times n}$, and let b be a given vector in $\mathbf{R}^n$. We address the problem of finding a unique vector x^* that fits the algebraic linear equation given by

$$Ax = b, \quad x \in \mathbf{R}^n. \tag{1}$$

To solve this linear system, the matrix A is split into two matrices M and N such that $A = M - N$, with M being a nonsingular matrix. As an example, the Jacobi method consists of choosing $M = D$ and $N = -(L + U)$, where D denotes the diagonal of A, U its strict upper part, and L its strict lower part. Equation (1) can then be rewritten in the form $x = M^{-1}Nx + M^{-1}b$, so by setting $T = M^{-1}N$ and $c = M^{-1}b$, we obtain an equivalent equation given by $x = Tx + c$. Let f be the function defined by

$$\begin{aligned} f : \mathbf{R}^n &\to \mathbf{R}^n \\ x &\mapsto Tx + c \end{aligned}$$

It is then obvious that the problem (1) can be reformulated as the problem of finding a unique fixed point of f. Given an initial vector $x(0) \in \mathbf{R}^n$, we deduce an iterative method given by the relation

$$x(k + 1) = f(x(k)), \quad \forall k \in \mathbf{N}. \tag{2}$$

From a mathematical point of view, beyond ensuring that the iterative method converges to the solution of the problem, a suitable criterion must be defined so that the convergence could be numerically detected. At the algorithmic level, considering a parallel execution, this detection must be performed using distributed data, in the most efficient way. We will see that asynchronous iterations introduce an additional complexity to the convergence detection issue. Let us now specify the definition of the convergence of an iterative method. Relatively to Eq. (1), an iterative method generating a sequence $\{y(k)\}_{k\in\mathbf{N}}$ of vectors in $\mathbf{R}^n$ is said to be convergent if and only if

$$\forall y(0) \in \mathbf{R}^n, \quad \lim_{k\to+\infty} y(k) = x^*.$$

In other words, for any $x(0) \in \mathbf{R}^n$, the method defined in (2) converges to x^* if and only if there exists a vector norm $\|.\|$ such that

$$\lim_{k\to+\infty} \|x(k) - x^*\| = 0. \tag{3}$$

It is proved (see [3], for instance) that such condition is equivalent to having the spectral radius of T, denoted by $\rho(T)$, satisfying the condition $\rho(T) < 1$, which also equals the existence of a matrix norm $\|.\|$ such that

$$\|T\| < 1. \tag{4}$$

Then a practical approach consists of ensuring (4) for a specific given norm $\|.\|_p$, $p \geq 1$, so that, from (3), a termination criterion can be specified by a relation of the form

$$\|f(x(k)) - x(k)\|_p \simeq 0, \quad k \in \mathbf{N}$$

for instance.

2.2 Synchronous parallel iterations

Let m be an integer in $\mathbf{N}^*$, with $m \leq n$. Let also $n_1, n_2, \ldots, n_m$ be m integers in $\mathbf{N}^*$ such that

$$\mathbf{R}^n = \mathbf{R}^{n_1} \times \cdots \times \mathbf{R}^{n_m}$$

so that any vector $x \in \mathbf{R}^n$ can be decomposed as $x = (x_1, \ldots, x_m)$. Now we consider f as a general function of the form

$$\begin{aligned} f : \mathbf{R}^n &\to \mathbf{R}^n \\ x &\mapsto f(x) \end{aligned}$$

and let

$$f_i : \mathbf{R}^{n_1} \times \cdots \times \mathbf{R}^{n_m} \to \mathbf{R}^{n_i}, \quad i \in \{1, 2, \ldots, m\}$$

be m functions defined such that we have

$$\forall x \in \mathbf{R}^n, \quad f(x) = (f_1((x_1, \ldots, x_m)), \ldots, f_m((x_1, \ldots, x_m))). \tag{5}$$

Given an initial vector $(x_1(0), \ldots, x_m(0))$ in $\mathbf{R}^{n_1} \times \cdots \times \mathbf{R}^{n_m}$, we consider a synchronous parallel iterative method given by the relation

$$x_i(k+1) = f_i((x_1(k), \ldots, x_m(k))), \qquad \forall i \in \{1, \ldots, m\}, \quad \forall k \in \mathbf{N}. \tag{6}$$

Then, according to (5), we would obviously have an equivalence between (2) and (6), which means that the convergence properties of the iterative method remain unaltered.

Let us consider a set of m processes. Two algorithmic models can be applied for implementing the synchronous parallel iterative method. The most trivial scheme is the Synchronous Iterations with Synchronous Communications (SISC), as described by Algorithm 1 and illustrated in Fig. 1a. Here, computation and communication are two totally distinct phases. For each iteration, computation is performed first, then messages are sent and received. We notice in Fig. 1a that any process willing to send information to a slower one needs to wait until the latter finishes its computation and requests the reception of the message. When an homogeneous machine with fast network connection, and a good load balancing is used, the wasted time is small. But with the evolution of the hardware, the efficiency of the SISC scheme is decreasing. Indeed the bandwidth and the latency of network connections are less improved than the processing power. In addition, the use of cheaper alternatives to supercomputers, like clusters or grids, amplifies the problem since interconnections in these systems are relatively slow.

Algorithm 1: SISC scheme on a process i

```
1  $k \leftarrow 0$
2  repeat
3      $x_i(k+1) \leftarrow f_i((x_1(k), \ldots, x_m(k)))$
4      foreach $j \in \{1, \cdots, i-1, i+1, \cdots, m\}$ do
5          Request the reception of $x_j(k+1)$ from process $j$
6          Request the sending of $x_i(k+1)$ to process $j$
7      end
8      Wait for communications to complete
9      $k \leftarrow k+1$
10 until $\|x(k) - x(k-1)\|_p \simeq 0$;
```

A well-known improvement consists of overlapping communication by computation. In this scheme, called Synchronous Iterations with Asynchronous Communications (SIAC), a process starts sending data immediately after its computation, even if the destination process is still computing. An illustration is given in Fig. 1c. As one can notice in Algorithm 2, "asynchronous communications" refers to the asynchronism between computation and communication, and not to an underlying asynchronous communication layer. Here, receptions start at the beginning of the iteration rather than at the end of the computation. This usually requires a little more memory, since the buffers must be duplicated, one being used for data reception, and another one for computation. Figure 1c does not show much improvement compared to Fig. 1a, but in real-world cases, the global resolution time can be consistently decreased, depending on both the network latency and the size of the messages. Indeed, with such overlapping, there is almost no more time dedicated to communication on slowest processes. On another hand, if the processes are perfectly balanced, there will be no overlap at all, and this scheme will not reduce the total execution time. It could also be possible to send parts of data as soon as they are computed. Ideally, most of the messages are thus sent and received before the end of the iteration. This way, dedicated communication time between two iterations could be reduced even on the fastest processes. Such overlapping is thus particularly interesting for an homogeneous set of processors.

Algorithm 2: SIAC scheme on a process i

```
1  $k \leftarrow 0$
2  repeat
3      foreach $j \in \{1, \cdots, i-1, i+1, \cdots, m\}$ do
4          Request the reception of $x_j(k+1)$ from process $j$
5      end
6      $x_i(k+1) \leftarrow f_i((x_1(k), \ldots, x_m(k)))$
7      foreach $j \in \{1, \cdots, i-1, i+1, \cdots, m\}$ do
8          Request the sending of $x_i(k+1)$ to process $j$
9      end
10     Wait for receptions to complete
11     $k \leftarrow k+1$
12 until $\|x(k) - x(k-1)\|_p \simeq 0$;
```

While coupling SIAC scheme and dynamic load balancing may produce very efficient implementations of iterative methods, the high frequency of synchronization between iterations remains an important limitation. Asynchronous iterations have thus been studied since 1967 for accelerating the parallel execution of some classes of iterative methods [15,20,34,37].

2.3 Asynchronous parallel iterations

Let $\{P(t)\}_{t\in\mathbf{N}}$ be a sequence such that

$$\forall t \in \mathbf{N}, \quad \begin{cases} P(t) \subseteq \{1, \ldots, m\} \\ P(t) \neq \emptyset \end{cases}$$

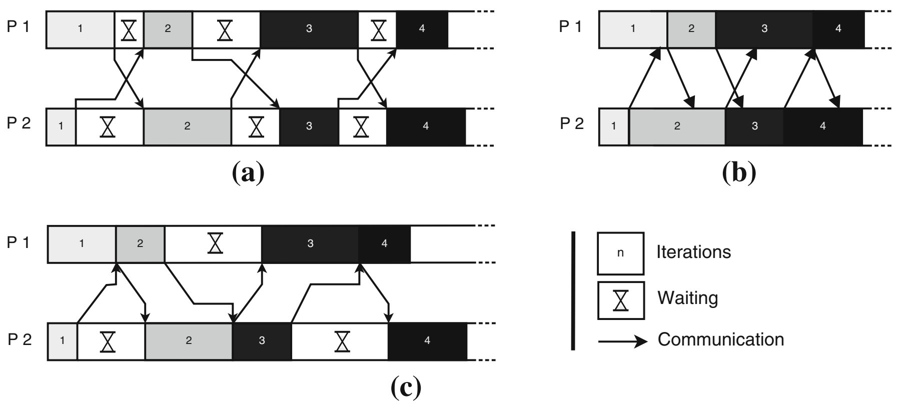

Fig. 1 Parallel iterative algorithms schemes. *P* process

The relation in (6) is then generalized to

$$\begin{cases} x_i(t+1) = f_i((x_1^{(i)}(t), \dots, x_i(t), \dots, x_m^{(i)}(t))), & \forall i \in P(t) \\ x_i(t+1) = x_i(t), & \forall i \notin P(t) \end{cases}, \quad \forall t \in \mathbf{N} \tag{7}$$

where for any i in $P(t)$, we have

$$x_j^{(i)}(t) = x_j(\tau_j^{(i)}(t)), \qquad \tau_j^{(i)}(t) \in \{0, 1, \dots, t\}, \quad j \in \{1, \dots, m\} \tag{8}$$

(7) and (8) define the Asynchronous Iterations with Asynchronous Communications (AIAC) scheme. Supposing that there is a set of m processes, the notation “$x_j^{(i)}(t)$” may stand for “the version of the component j of the vector x accessed by the process i at the *global iteration* $t+1$”. The Eq. (8) indicates that this version can be the value computed by the process j at any global iteration anterior to t. This model is then completed by the specification of two major assumptions given by

$$\begin{cases} \forall i \in \{1, \dots, m\}, & \text{card}\ \{t \in \mathbf{N} | i \in P(t)\} = +\infty \\ \forall i, j \in \{1, \dots, m\}, & \lim_{t \to +\infty} \tau_j^{(i)}(t) = +\infty \end{cases}$$

In other words, any component of the iterative solution vector is always eventually updated using updated values of all other components.

On the algorithmic side, the general principle is quite simple: when a process finishes its current iteration, it starts the next one without waiting. If it has not received new data from the other processes, it simply uses the last values that it received. Figure 1b represents an execution of the AIAC scheme, also described by Algorithm 3. AIAC algorithms usually need more iterations to attain the same accuracy than their synchronous counterparts, hence they are not a priori faster, relatively to the execution time. Several parameters related to the computing environment have to be taken into account when switching from synchronous to asynchronous iterations. It is thus quite hard to accurately forecast the benefit from the asynchronism, and in some cases, partial synchronization could lead to better results. For instance, a way to keep a good convergence speed is to make groups of processor cores using synchronous iterations and to resort to asynchronous iterations between these groups. This well suits most of the current systems which have a hierarchical architecture. Processor cores share a cache memory, allowing very fast synchronization, and nodes are interconnected through a potentially hierarchical network which is relatively slow. A group could, however, be broken at any time during the execution, switching its cores to an AIAC scheme.

Algorithm 3: AIAC scheme on a process i

```
1  $k_i \leftarrow 0$
2  repeat
3      foreach $j \in \{1, \cdots, i-1, i+1, \cdots, m\}$ do
4          Request the reception of $x_j(k_j)$ from process $j$
5      end
6      $x_i(k_i + 1) \leftarrow f_i((x_1(k_1), \ldots, x_m(k_m)))$
7      foreach $j \in \{1, \cdots, i-1, i+1, \cdots, m\}$ do
8          Request the sending of $x_i(k_i + 1)$ to process $j$
9      end
10     $k_i \leftarrow k_i + 1$
11 until $\|f((x_1(k_1), \ldots, x_m(k_m))) - (x_1(k_1), \ldots, x_m(k_m))\|_p \simeq 0$;
```

While Algorithm 3 gives a very simplified description of the AIAC scheme, its implementation obviously raises non-trivial questions. Globally, these questions are related to the management of the successive communication requests, the management of the communication buffers and the termination detection. Here, we propose a new communication library that handles all these questions, while allowing programmers to write only one source code for executing either an SIAC or an AIAC scheme. The choice can then be made at runtime. Additionally, synchronous and asynchronous iterations can be automatically mixed, based on the topology of the network and on the characteristics of the computing nodes. While our library is written in C++, it also provides a C language API very similar to the MPI one. This makes the library very easy to be used.

3 MPI-based communication library for iterative algorithms

On one hand, we saw that asynchronous formulation of iterations is a generalization of the synchronous one. Furthermore, we can notice, indeed, that both the SIAC and the AIAC schemes globally feature the same algorithmic pattern. It is thus possible to provide a unique API for implementing both asynchronous and synchronous iterations, by considering a parameter that would define the appropriate behavior of the communication library. On other hand, the usability of such a library would be to make it possible to implement this algorithmic pattern as is. Considering these aspects, the template of an iterative algorithm implemented with JACK is given by Algorithm 4. Although we present only some particular generic functionalities for common iterative algorithms, the JACK library provides a more complete set of functions that includes most of the MPI specification. It can thus be easily used as a whole by replacing each MPI call.

Algorithm 4: JACK-based SIAC/AIAC scheme on a process i

```
1  JACK::Setup(JACK::ASYNCHRONOUS) //*
2  [h]or JACK::Setup(JACK::SYNCHRONOUS)
3  JACK::SetupBufferManager($n_i$)
4  foreach $j \in \{1, \cdots, i-1, i+1, \cdots, m\}$ do
5      JACK::StartContinuousRecv($n_j$, $j$, $x_j(k_j)$)
6  end
7  $k_i \leftarrow 0$
8  repeat
9      JACK::CreateBuffer($x_i(k_i+1)$)
10     $x_i(k_i+1) \leftarrow f_i((x_1(k_1), \ldots, x_m(k_m)))$
11     foreach $j \in \{1, \cdots, i-1, i+1, \cdots, m\}$ do
12         JACK::Send($x_i(k_i+1)$, $n_i$, $j$)
13     end
14     localConvergence $\leftarrow (\|x_i(k_i+1) - x_i(k_i)\|_\infty \simeq 0)$
15     globalConvergence $\leftarrow$ JACK::LocalCvToGlobalCv(localConvergence)
16     foreach $j \in \{1, \cdots, i-1, i+1, \cdots, m\}$ do
17         $x_j(k_j) \leftarrow$ JACK::TestOrWaitForCompletion($j$)
18     end
19     $k_i \leftarrow k_i + 1$
20 until globalConvergence;
```

3.1 Overview of the API

Initialization JACK provides a communicator object (`Jack::Communicator`) that defaults to the SIAC mode (`Jack::c_SYNCHRONOUS`). We thus have to specifically switch to the AIAC mode (`Jack::c_ASYNCHRONOUS`) when needed. It is possible to switch back to the SIAC mode during computation, however, being aware of possible consistency issues. A buffer manager (`Jack::BufferMgr`) can be used to avoid handling the buffering of successive outgoing messages. Indeed, the number of buffers involved in communication at a given time could be nondeterministic. We highlight the fact that resorting to the buffer manager does not imply additional JACK calls dedicated to memory freeing. The buffer manager is initialized with the size of the messages that will be sent. Listing 1 snapshots the C++ source code for this initialization part, according to Algorithm 4.

Listing 1 Initialization

```
Jack::Communicator comm = Jack::GetCommWorld();
comm = comm.Dup(Jack::c_ASYNCHRONOUS);
Jack::BufferMgr bufMgr;
bufMgr.Setup<double>(n[i]);
```

Iterations JACK provides a particular message reception request (`Jack::ContinuousRequest`) that automatically matches every new incoming messages on a channel. The request is submitted once before the iteration loop, and returns a reception buffer (`Jack::Buffer`) that can be accessed for the computation (See Listing 2). Then users do not need to manage successive reception requests. Additionally, the multiple buffering is handled automatically. While this could be sufficient for a pure AIAC algorithm, the SIAC scheme requires a request completion function after the computation. We thus resort to a generic function that either checks or waits for the completion of a continuous request (see Listing 3). Note, however, that, by definition, a continuous request is never completed. The completion that we refer to is thus related to the last automatic underlying request. In synchronous mode, it behaves like the MPI Wait function. In asynchronous mode, it checks if some data has been received since the last call. In this case, a parameter buffer is updated, using last received message. The buffer can thus be safely accessed until the next call to the test function. For both synchronous and asynchronous modes, there is no need to wait for the completion of message sending requests. Yet, attention must be paid to the validity of the data being sent, as they could be altered at the same time by ongoing computation. A buffer can be provided by JACK at each iteration, and each buffer is automatically freed once its contents sending completes. See Listing 4 for the corresponding C++ source code, according to Algorithm 4. Listing 5 shows how JACK buffers could be accessed for the computation part.

Listing 2 Reception request

```
int tag = 0; // MPI meaning
bool forceSynchronous = true;
Jack::ContinuousRequest * recvRq;
Jack::Buffer * recvBuf;
recvBuf = new Jack::Buffer[m];
recvRq = new Jack::ContinuousRequest[m];
for (int j = 0; j < m; j++) {
    if (j != i) {
      recvBuf[j] = comm.StartContinuousRecv<double>(n[j], j, tag, recvRq[j],
            !forceSynchronous);
    }
}
```

Listing 3 Reception buffer update

```
Jack::Status status;
for (int j = 0; j < m; j++) {
    if (j != i) {
        recvRq[j].TestOrWaitForCompletion(recvBuf[j], &status);
    }
}
```

Listing 4 Message sending

```
Jack::Buffer sendBuf = bufMgr.GetBuffer();
for (int j = 0; j < m; j++) {
    if (j != i) {
        Jack::Request sendRq; // automatically freed
        comm.Send(sendBuf, j, tag, sendRq);
    }
}
```

Listing 5 Accessing buffers

```
double ** x;
double * x_i;
x = new (double*)[m];
x[i] = new double[n[i]];
for (int j = 0; j < m; j++) {
    if (j != i) {
        x[j] = recvBuf[j].GetData();
    }
}
x_i = sendBuf.GetData();
Compute(x_i, x); // x_i <-- f_i(x)
```

Termination Algorithm 3 clearly points out the problem of detecting the convergence of asynchronous iterations. Indeed, let us consider the condition

$$\|f((x_1(k_1), \dots, x_m(k_m))) - (x_1(k_1), \dots, x_m(k_m))\|_p \simeq 0.$$

On one hand, the vectors $(x_1(k_1), \dots, x_m(k_m))$ are probably different in each process i. On the other hand, applying the whole function f to a given vector would require a synchronization, what is not an option in an AIAC scheme. The problem of terminating asynchronous iterations has been quite investigated since 1989 [8,22,35]. The main approach leads to a modification of the mathematical model (presented in Sect. 2.3) such that the resulting algorithmic pattern fits a classical distributed termination model [19,29,33]. A second approach consists of implementing a monitoring algorithm that collects and evaluates some information about the ongoing iterative computation.

Then actually, a convergence detection tool would be the most useful feature of a communication library designed for asynchronous iterative computing. To minimize the dependency between the convergence detection and the iterative method, we considered the second approach and provide an implementation of the method proposed in [4]. It is based on a leader election algorithm in tree topology (see [25, Sect. 4.4.3]) wherein the authors introduced messages that cancel notifications of local convergence. An important parameter thus need to be tuned for defining a period after which, in absence of cancellation message, a convergence state (global or local) is validated.

One can notice by Algorithm 4 that the convergence criterion

$$\|f(x(k)) - x(k)\|_p \simeq 0, \quad p \geq 1$$

is restricted to

$$\|f(x(k)) - x(k)\|_\infty \simeq 0$$

Indeed, it is assumed that the global convergence derived from the local convergence of all the processes. Such assumption is merely based on the definition of the maximum norm, which is given by

$$\|x\|_\infty = \max_{1\leq i\leq n} |x_i|, \quad x \in \mathbf{R}^n$$

so that we also have

$$\|x\|_\infty = \max_{1\leq i\leq m} \|x_i\|_\infty, \quad x \in \mathbf{R}^{n_1} \times \cdots \times \mathbf{R}^{n_m}.$$

Listing 6 shows how to use the convergence detection tool provided by JACK, according to Algorithm 4.

Listing 6 Convergence detection

```
Jack::ConvergenceDetector convDetector = comm.CreateCvDetector(tag+1);
double norm = MaxNorm(x_i, x[i]); // maximum norm of (x_i - x[i])
bool localConv = (norm < 10e-9);
bool globalConv = convDetector.LocalCvToGlobalCv(localConv);
```

3.2 Communication requests

Message sending In the JACK library, a channel is identified by a source, a destination and a tag. In asynchronous mode, before querying the sending of a message on a channel, the state of this channel is checked. If there already exists an uncompleted query, the new message sending request is ignored. Indeed, busy channels indicate that messages are generated faster than they can be sent. This avoids unnecessary queuing, since in asynchronous methods, only the newest data are really useful to the destination process.

It would also be possible to only queue the last message until the previous one is fully transmitted. Then any new message to be sent takes the place of the second element in the queue. This could, however, keep the communication channel always busy.

At last, message sending cancellation would not be a suitable approach. In the case where the time to send a message is always longer than the time to generate it, all the requests would be canceled before completion. Furthermore, canceling a request could be a very expensive operation, which may not even be implemented in some MPI libraries.

Message reception Data are received in a temporary buffer and when needed, delivered in the computation buffer (passed as parameter). To avoid copy in the case where there is only one pending message (mostly in synchronous mode), a first reception is directly operated into the computation buffer. With MPI libraries, a new reception must be requested as soon as a message is received.

It would also be possible to maintain a set of pairs of buffers and requests, according to an expected number of receptions in one iteration. Then the library would automatically switch from one buffer to another as the requests complete. With enough buffers, one could wait until the end of each iteration before restarting the requests. In all cases, JACK thus needs to continuously execute some updating codes.

Continuous update Tools like convergence detector and continuous request are really effective if JACK can regularly execute some portions of code. It thus needs to be invoked on a regular basis, to restart reception requests, to handle or send convergence messages, and so on. While these 'continuous' updates are not absolutely necessary for asynchronous reception, they can be useful for other various components of JACK, by reducing the latency of some operations. The most trivial solution has been to use any call to JACK as an opportunity to make updates. This implies that when the user calls a JACK function, the time since the last call is checked and if it is bigger than a threshold, then an updating function is called before executing the expected command. The overhead of this approach is minimal and it does not need any change in the iterative algorithm pattern. It is, however, allowed to directly invoke the JACK update function itself.

As another possibility, JACK can also be configured to use kernel threads, preventing users from handling multi-threading issues. This is particularly interesting for programs with long running computational parts without JACK calls. Indeed, the JACK kernel thread can automatically interrupt the computation, for executing update codes. One thing to configure is the frequency at which the computation will be interrupted. On Linux OS (Operating Systems), the computation thread can be given a low priority, which increases the frequency of the interruptions. The default behavior for a thread is to run 100 ms before being interrupted, which can be reduced to 5 ms. On Windows, the JACK kernel thread is given a higher priority and is made to sleep for a desired time. When a sleeping thread with high priority wakes up, it interrupts the other threads. This multi-threading option, however, introduces some non-negligible overhead, and while most OS support the functionality, some specific High-Performance Computing (HPC) OS do not.

Back-end features Globally, the JACK library provides generic functionalities targeting both asynchronous and synchronous iterative algorithms in particular. These are features mostly related to convergence detection and continuous request. Obviously, they require some basic communication tools, more common to a larger set of distributed applications. An MPI-like layer of functions is thus provided as well, for both users and JACK complex tools. Actually, these functions are built upon communication interfaces, currently one for processes and another one for threads. JACK can mix inter-process and inter-thread communication, using the former interface only between processes on different computing nodes. When, however, a multi-thread application restricts JACK to the only MPI interface, but links with an MPI library that does not support multi-threading, a mutual exclusion mechanism ensures that all MPI calls are serialized. Such a layered architecture is also intended to ease the future integration of other interfaces like, for instance, the user Direct Access Programming Language (uDAPL) for RDMA support.

3.3 Convergence detection

With the AIAC scheme, the local condition, $\|x_i(k_i + 1) - x_i(k_i)\|_\infty \simeq 0$, may be satisfied at a given iteration k_i, and then no longer holds after the process i receives new

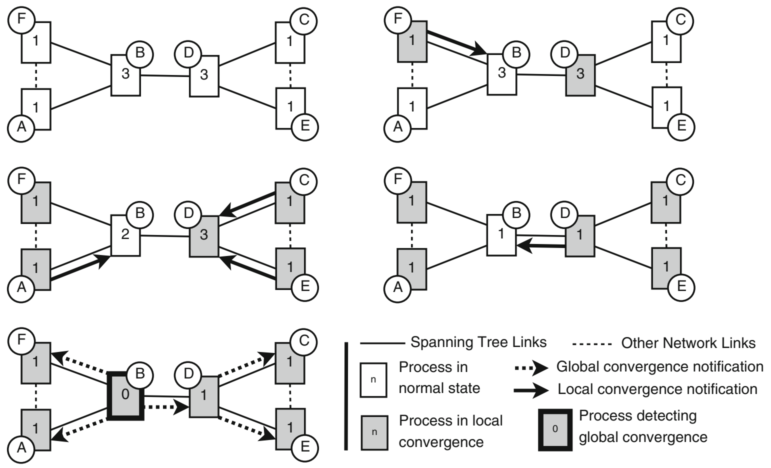
F
A
B
D
C
E
Spanning Tree Links
Other Network Links
Process in normal state
Process in local convergence
Global convergence notification
Local convergence notification
Process detecting global convergence

Fig. 2 Leader election example. The *number inside each box* is the number of neighbors from which the process has not received any convergence notification. Step 1: processes *F* and *D* reach convergence, but only *F* can send notification. Step 2: *A*, *C* and *E* reach convergence, then send notifications. *B* receives notification from *F*, then updates its remaining number of notifications. Step 3: *D* sends notification to its only neighbor from which it did not receive any notification. Step 4: *B* reaches convergence, then detects and notifies global convergence

data from its neighbors in the communication graph. A period of confirmation is thus required, for receiving at least one update from each neighbor, so that the process can observe a persistent local convergence. While this does not guarantee definitive local convergence, it reduces the amount of useless convergence notifications. As previously seen, convergence states are notified on the network through a leader election protocol that operates on a spanning tree over the communication graph. Algorithms for building a spanning tree can be found in [13,31]. Here we recall, for each process, the rules that describe the leader election protocol, to which we associate convergence notification. The rules are enabled only when local convergence is reached. Divergence notification through cancellation messages only introduce consistency issues which are further detailed in [4]. Figure 2 illustrates the basic protocol on a network of five processes.

Rule 1: when local convergence notification is received from all neighbors (in the tree), global convergence is notified to all of them.
Rule 2: when there remains only one neighbor from which no notification has been received, local convergence is notified to that neighbor.
Rule 3: when global convergence notification is received from a neighbor, global convergence is notified to all other neighbors.

4 Numerical experiments

Using the JACK library, we experimented different implementations of the Jacobi method (see Sect. 2.1), following two different models of parallel iterations.

Row-band parallel scheme Let the matrix T be decomposed into sub-matrices T_i, $i \in \{1, \dots, m\}$, each T_i being an $n_i \times n$ real matrix. Considering m processes, we define functions f_i by

$$f_i((x_1, \dots, x_m)) = T_i(x_1, \dots, x_m) + c_i.$$

Then, given an initial vector $(x_1(0), \dots, x_m(0))$ in $\mathbf{R}^{n_1} \times \cdots \times \mathbf{R}^{n_m}$, the row-band parallel scheme is given by the relation

$$x_i(k+1) = f_i((x_1(k), \dots, x_m(k))), \qquad \forall i \in \{1, \dots, m\}, \quad \forall k \in \mathbf{N}.$$

Additionally, the sparsity of each T_i is taken into account for optimizing the exchange of the components x_j, $j \neq i$. Communication occurs for only subcomponents of x_j for which the corresponding coefficient in T_i is a nonzero value.

Sub-structuring parallel scheme After a decomposition of the domain into $(m-1)$ sub-structures, a classical renumbering, as introduced in [26,32], leads to

$$A = \begin{pmatrix} A_{11} & 0 & \cdots & 0 & A_{1m} \\ 0 & A_{22} & \ddots & \vdots & A_{2m} \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & A_{(m-1)(m-1)} & A_{(m-1)m} \\ A_{m1} & A_{m2} & \cdots & A_{m(m-1)} & A_{mm} \end{pmatrix}, \qquad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{m-1} \\ b_m \end{pmatrix}$$

so that the mth components represent the interface between the sub-structures [16]. A and b are then distributed on sub-structures $i = 1, \dots, m-1$ as follows:

$$A^{(i)} = \begin{pmatrix} A_{ii} & A_{im} \\ A_{mi} & A_{mm}^{(i)} \end{pmatrix}, \qquad b^{(i)} = \begin{pmatrix} b_i \\ b_m^{(i)} \end{pmatrix}$$

with

$$\sum_{i=1}^{m-1} A_{mm}^{(i)} = A_{mm}, \qquad \sum_{i=1}^{m-1} b_m^{(i)} = b_m$$

Equivalently, we also have

$$T^{(i)} = \begin{pmatrix} T_{ii} & T_{im} \\ T_{mi} & T_{mm}^{(i)} \end{pmatrix}, \qquad c^{(i)} = \begin{pmatrix} c_i \\ c_m^{(i)} \end{pmatrix}$$

with

$$T_{ii} = M_{ii}^{-1} N_{ii}, \quad T_{im} = M_{ii}^{-1} N_{im}, \quad T_{mi} = M_{mm}^{-1} N_{mi}, \quad T_{mm}^{(i)} = M_{mm}^{-1} N_{mm}^{(i)}$$

and

$$\sum_{i=1}^{m-1} T_{mm}^{(i)} = T_{mm}, \qquad \sum_{i=1}^{m-1} c_m^{(i)} = c_m.$$

We define the following local vectors:

$$y_i = \begin{pmatrix} x_i \\ x_m^{(i)} \end{pmatrix}, \quad \forall i \in \{1, \ldots, m-1\}$$

with

$$\sum_{i=1}^{m-1} x_m^{(i)} = x_m$$

and the global vector $y = (y_1, y_2, \ldots, y_{m-1})$. Considering $m-1$ processes, we finally define functions g_i by

$$g_i(y) = T^{(i)} \begin{pmatrix} x_i \\ \sum_{j=1}^{m-1} x_m^{(j)} \end{pmatrix} + c_i, \qquad \forall i \in \{1, \ldots, m-1\}.$$

Then, given an initial vector $y(0) = (y_1(0), y_2(0), \ldots, y_{m-1}(0))$ in $(\mathbf{R}^{n_1} \times \mathbf{R}^{n_m}) \times (\mathbf{R}^{n_2} \times \mathbf{R}^{n_m}) \times \cdots \times (\mathbf{R}^{n_{m-1}} \times \mathbf{R}^{n_m})$, the sub-structuring parallel implementation is given by the relation

$$y_i(k+1) = g_i((y_1(k), \ldots, y_{m-1}(k))), \qquad \forall i \in \{1, \ldots, m-1\}, \quad \forall k \in \mathbf{N}$$

Here as well, communication toward a process i is restricted to only subcomponents of y_j, $j \neq i$, that are required for an evaluation of g_i.

Problem and experimental settings We consider the equation $-\Delta u = h$, with h being a given function. We have implemented the synchronous row-band parallel solver, the synchronous sub-structuring [32] parallel solver and asynchronous sub-structuring [28] parallel solver within Alinea [27]. Alinea [27] is an advanced linear algebra library we have developed for massively parallel computations. The synchronous and asynchronous communications are managed by JACK, the new library presented in this paper. The computational platform is a cluster of 6 workstations on a 10 Mb Ethernet network. Each workstation consists of 2 quad-core Intel Xeon CPUs (Central Processing Units) at 2.67 GHz, 8-GB RAM, and a 64-bit Linux OS. A total of 48 processor cores are then provided.

A first domain Ω consists of a regular hexahedron, with homogeneous Dirichlet boundary conditions equal to zero on the boundary. This domain is meshed into a structured grid composed of 32,768 hexahedral elements and 35,937 nodes. A refinement produced a second mesh composed of 262,144 elements and 274,625 nodes.

Lagrange P1 finite elements discretization leads to symmetric matrices, respectively, of sizes 35,937 and 274,625.

A second domain Ω' consists in a flattened hexahedron, with homogeneous Dirichlet boundary conditions equal to zero on the vertical faces (normals perpendicular to the z-axis). This domain is also meshed into a structured grid composed of 60,000 hexahedral elements and 71,407 nodes. Again, a refinement produced a second mesh composed of 480,000 elements and 525,213 nodes. Lagrange P1 finite elements discretization leads to symmetric matrices, respectively, of sizes 71,407 and 525,213.

At last, we set a convergence criterion such that the solution $\tilde{x}$ satisfies the condition $\|f(\tilde{x}) - \tilde{x}\|_\infty \leq 10^{-6}$. Sequential execution times are reported in Table 1.

Numerical results By comparing Tables 2 and 3, we observe that in the case of the domain Ω, the sub-structuring method eventually outperforms the optimized row-band approach as the size of the matrix increases. On another hand, the results for the asynchronous sub-structuring implementation turned out to be unsteady for the

Table 1 Results from sequential executions

Case	Matrix size	Iterations	Time (s)
Ω	35,937	746	10.9
	274,625	2216	238.4
Ω'	71,407	6234	159.5
	525,213	13,791	2713.0

Table 2 Results on domain Ω, matrix of size 35,937

#Proc.	Sync. row-band		Sync. sub-struct.		Async. sub-struct.	
	T (s)	E (%)	T (s)	E (%)	T (s)	E (%)
16	2.5	26.94	2.6	25.88	2.2	31.74
32	2.3	14.84	2.4	14.46	1.7	19.96
48	2.7	8.43	2.6	8.70	1.8	12.47
64	2.1	8.13	2.4	7.02	1.6	10.64

T time, E efficiency

Table 3 Results on domain Ω, matrix of size 274,625

#Proc.	Sync. row-band		Sync. sub-struct.		Async. sub-struct.	
	T (s)	E (%)	T (s)	E (%)	T (s)	E (%)
16	37.1	40.21	32.1	46.35	51.5	28.91
32	28.7	25.98	24.6	30.25	31.2	23.84
48	33.6	14.79	25.8	19.27	43.7	11.37
64	28.0	13.31	23.0	16.18	18.0	20.66

T time, E efficiency

Table 4 Results on domain Ω', matrix of size 71,407

#Proc.	Sync. row-band		Sync. sub-struct.		Async. sub-struct.	
	T (s)	E (%)	T (s)	E (%)	T (s)	E (%)
16	133.1	7.49	22.9	43.58	32.7	30.52
32	136.6	3.65	23.7	21.02	20.6	24.17
48	151.7	2.19	22.9	14.48	21.9	15.19
64	149.2	1.67	22.6	11.02	16.8	14.80

T time, E efficiency

Table 5 Results on domain Ω', matrix of size 525,213

#Proc.	Sync. row-band		Sync. sub-struct.		Async. sub-struct.	
	T (s)	E (%)	T (s)	E (%)	T (s)	E (%)
16	7,399.3	2.29	273.7	61.95	243.8	69.55
32	8,429.6	1.01	228.3	37.13	350.3	24.20
48	10,769.9	0.52	191.9	29.46	148.1	38.16
64	11,921.9	0.36	164.6	25.76	100.9	42.01

T time, E efficiency

larger matrix on Ω. Nevertheless, this method featured the best efficiency when up to 64 processes were used for the resolution. On the contrary, and quite unexpectedly, it outperformed the other methods on the smaller matrix, from only 16 processes. Tables 4 and 5 show that, on the domain Ω', the performance of the synchronous row-band method drastically drops, regardless both the size of the matrix and the number of processes. Here, for each matrix, the asynchronous approach is globally confirmed to be more efficient and more scalable than the synchronous one.

5 Conclusion

We investigated the development of JACK, a new MPI-based communication library targeting, but not limited to, asynchronous iterative algorithms. The main features were designed and presented according to fixed-point methods, which are the native mathematical framework of such algorithms. Users can simply resort to specialized tools proposed for asynchronous iterative computing, such as continuous reception request and convergence detector. An MPI-like layer of functions is available as well, more generally for other classes of distributed algorithms. While this library mainly requires MPI middleware, it is designed to easily integrate additional interfaces for various other communication libraries.

JACK was here experimented for the solution of linear problems arising from finite element methods. We considered one sequential and three JACK-based parallel implementations of the Jacobi iterative method, namely a synchronous optimized row-band parallelization, a synchronous sub-structuring parallel scheme, and an asynchronous version of the sub-structuring approach. Simulations were run on two different geometries and different configurations. As expected, the parallel sub-structuring approach was clearly far more efficient than the row-band parallelization, despite a sparsity-based optimization which reduces the amount of inter-process exchanges for the latter method. More important, if we consider, for a given size of problem, the number of processor cores from which there is no more gain on a synchronous parallel execution time, we notice that asynchronous iterations allow us to use more cores and keep reducing the execution time. In a general way, we can confirm, using the JACK library, that asynchronous iterations increase both the efficiency and the scalability of the sub-structuring method.

References

1. Bahi J, Domas S, Mazouzi K (2004) Jace: a java environment for distributed asynchronous iterative computations. In: Proceedings. 12th euromicro conference on parallel, distributed and network-based processing, 2004, pp 350–357
2. Bahi J, Miellou JC, Rhofir K (1997) Asynchronous multisplitting methods for nonlinear fixed point problems. Numer Algorithms 15(3–4):315–345
3. Bahi JM, Contassot-Vivier S, Couturier R (2007) Parallel iterative algorithms: from sequential to grid computing. In: Numerical analysis and scientific computing series. Chapman & Hall/CRC, UK
4. Bahi JM, Contassot-Vivier S, Couturier R, Vernier F (2005) A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms. IEEE Trans Parallel Distrib Syst 16(1):4–13
5. Bahi JM, Couturier R, Vuillemin P (2007) Jacev: a programming and execution environment for asynchronous iterative computations on volatile nodes. In: High performance computing for computational science—VECPAR 2006. Lecture Notes in Computer Science, vol 4395. Springer Berlin, pp 79–92 (2007)
6. Baudet GM (1978) Asynchronous iterative methods for multiprocessors. J ACM 25(2):226–244
7. Bertsekas DP, Tsitsiklis JN (1989) Parallel and distributed computation: numerical methods. Prentice-Hall Inc, Upper Saddle River
8. Bertsekas DP, Tsitsiklis JN (1991) Some aspects of parallel and distributed iterative algorithmsa survey. Automatica 27(1):3–21
9. Bru R, Elsner L, Neumann M (1988) Models of parallel chaotic iteration methods. Linear Algebra Appl 103:175–192
10. Bru R, Migallon V, Penadés J (1995) Chaotic methods for the parallel solution of linear systems. International meeting on vector and parallel processing. Comput Syst Eng 6(4–5):385–390
11. Bru R, Migallón V, Penadés J, Szyld DB (1995) Parallel, synchronous and asynchronous two-stage multisplitting methods. Electron Trans Numer Anal 3:24–38
12. Chajakis ED, Zenios SA (1991) Synchronous and asynchronous implementations of relaxation algorithms for nonlinear network optimization. Parallel Comput 17(8):873–894
13. Chang EJH (1982) Echo algorithms: depth parallel operations on general graphs. IEEE Trans Softw Eng SE–8(4):391–401
14. Chau M, Couturier R, Bahi J, Spiteri P (2011) Parallel solution of the obstacle problem in grid environments. Int J High Perform Comput Appl 25(4):488–495
15. Chazan D, Miranker W (1969) Chaotic relaxation. Linear Algebra Appl 2(2):199–222
16. Cheik Ahamed A, Magoulès F (2014) Parallel sub-structuring methods for solving sparse linear systems on a cluster of gpus. In: Proceedings of the 16th international conference on high performance and communications (HPCC 2014), Paris, France, Aug. 20–22, 2014. IEEE Computer Society, New York, pp 121–128 (2014)
17. Couturier R, Domas S (2007) Crac: a grid environment to solve scientific applications with asynchronous iterative algorithms. In: Parallel and distributed processing symposium, 2007. IPDPS 2007. IEEE International, New York, pp 1–8
18. Deren W, Zhongzhi B, Evans DJ (1994) Asynchronous multisplitting relaxed iterations for weakly nonlinear systems. Int J Comput Math 54(1–2):57–76

19. Dijkstra EW, Feijen WHJ, van Gasteren AJM (1983) Derivation of a termination detection algorithm for distributed computations. Inf Process Lett 16(5):217–219
20. Donnelly JDP (1971) Periodic chaotic relaxation. Linear Algebra Appl 4(2):117–128
21. El Baz D, Spiteri P, Miellou JC, Gazen D (1996) Asynchronous iterative algorithms with flexible communication for nonlinear network flow problems. J Parallel Distrib Comput 38(1):1–15
22. Evans D, Chikohora S (1998) Convergence testing on a distributed network of processors. Int J Comput Math 70(2):357–378
23. Juliand J, Perrin GR, Spitéri P (1981) Simulations d'exécutions parallèles d'algorithmes numériques asynchrones. In: 1st conférence AMSE, Lyon
24. Lei L (1989) Convergence of asynchronous iteration with arbitrary splitting form. Linear Algebra Appl 113:119–127
25. Lynch NA (1996) Distributed algorithms. Morgan Kaufmann Publishers Inc., San Francisco
26. Maday Y, Magoulès F (2006) Absorbing interface conditions for domain decomposition methods: a general presentation. Comput Meth Appl Mech Eng 195(29–32):3880–3900
27. Magoulès F, Cheik Ahamed A (2015) Alinea: an advanced linear algebra library for massively parallel computations on graphics processing units. Int J High Perform Comput Appl 29(3):284–310
28. Magoulès F, Venet C (2016) Asynchronous iterative sub-structuring methods. J Math Computer Simul
29. Mattern F (1987) Algorithms for distributed termination detection. Distrib Comput 2(3):161–175
30. OLeary DP, White RE (1985) Multi-splittings of matrices and parallel solution of linear systems. SIAM J Algebraic Discrete Meth 6(4):630–640
31. Perlman R (1985) An algorithm for distributed computation of a spanningtree in an extended lan. SIGCOMM Comput Commun Rev 15(4):44–53
32. Przemieniecki JS (1963) Matrix structural analysis of substructures. Am Inst Aeronaut Astronaut J 1(1):138–147
33. Rana SP (1983) A distributed solution of the distributed termination problem. Inf Process Lett 17(1):43–46
34. Rosenfeld J (1967) A case study on programming for parallel processors. Tech. Rep. RC-64, IBM Thomas J. Watson Research Center, USA
35. Savari S, Bertsekas D (1996) Finite termination of asynchronous iterative algorithms. Parallel Comput 22(1):39–56
36. Spiteri P, Chau M (2002) Parallel asynchronous Richardson method for the solution of obstacle problem. In: Proceedings. 16th annual international symposium on high performance computing systems and applications, pp 133–138 (2002)
37. Tsitsiklis JN, Bertsekas DP, Athans M (1986) Distributed asynchronous deterministic and stochastic gradient optimization algorithms. IEEE Trans Autom Control 31(9):803–812