

Convergence Detection of Asynchronous Iterations Based on Modified Recursive Doubling

Qinmeng Zou, Frédéric Magoulès

▶ **To cite this version:**

Qinmeng Zou, Frédéric Magoulès. Convergence Detection of Asynchronous Iterations Based on Modified Recursive Doubling. 17th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES), Oct 2018, Wuxi, China. 10.1109/dcabes.2018.00081 . hal-01964246

HAL Id: hal-01964246

https://centralesupelec.hal.science/hal-01964246

Submitted on 1 Jul 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Convergence Detection of Asynchronous Iterations based on Modified Recursive Doubling

Qinmeng Zou* Frédéric Magoulès†

Abstract

This paper addresses the distributed convergence detection problem in asynchronous iterations. A modified recursive doubling algorithm is investigated in order to adapt to the non-power-of-two case. Some convergence detection algorithms are illustrated based on the reduction operation. Finally, a concluding discussion about the implementation and the applicability is presented.

Keywords: recursive doubling; asynchronous iterations; convergence detection; message passing interface

1 Introduction

Consider the following linear system

$$Ax = b,$$

where $A \in \mathbb{R}^{n\times n}$ and $b \in \mathbb{R}^n$. A splitting

$$A = M - N,$$

yields an iterative scheme

$$x^{k+1} = Tx^k + c,$$

where $k \in \mathbb{N}$, $T = M^{-1}N$ and $c = M^{-1}b$. Generally, this scheme is well suited for parallel computing of the form

$$x_i^{k+1} = T_i x^k + c_i, \quad i \in \{1, \ \ldots, \ p\},$$

where p is the number of processors. Here, x_i and c_i might be two values or two smaller vectors. Similarly, T_i might be a row vector or a smaller matrix. They are distributed in different processors. However, a specific point is required at the end of each iteration to synchronize between the processors. The waste of time may be significant in the case of unbalanced working load and node failure, which gives rise to the asynchronous iterative methods. The asynchronous iterative scheme has been proposed by Chazan and Miranker [4] for the solution of linear equations and generalized by several researchers (see, e.g., [17, 1, 5, 2]) for the general problem

$$x^{k+1} = f\left(x^k\right),$$

where f is a fixed point mapping. The asynchronous iterative scheme is shown as follows

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

where $\tau_{i,j,k} \leq k$ is a sequence of iterations with retards for each element j in each processor i, and $P^k \subset \{1, \ \ldots, \ p\}$ is a sequence of subsets of processor numbers. In this case, processors

*CentraleSupélec, Université Paris-Saclay, France.
†CentraleSupélec, Université Paris-Saclay, France (correspondence, frederic.magoules@hotmail.com).

are not required to wait for receiving all messages and allowed to keep on their own pace. We often add the following conditions to better investigate the chaotic process

$$\begin{cases} \left|\left\{k \in \mathbb{N} \mid i \in P^k\right\}\right| = +\infty, & \forall i \in \{1, \ldots, p\}, \\ \lim\limits_{k \to +\infty} \tau_{i,j,k} = +\infty, & \forall i, j \in \{1, \ldots, p\}, \end{cases}$$

where $|.|$ is the cardinality of a set that measures the number of elements. It means that no processors should be abandoned forever and more and more recent values should be used.

Asynchronous iterative algorithms must terminate after a finite number of iterations, as suggested in [19]. Thus, a practical implementation involves a set of admissible solutions S, such that

$$x^* \in S,$$

where x^* is a solution vector. We would like to find a vector $\bar{x}$ established by the components from each processor, and we have to evaluate $\bar{x} \in S$. If true, then $x^* = \bar{x}$; otherwise, continue the computation, as well as the evaluation. Thus, the termination condition can be expressed by a residual evaluation

$$\|f(\bar{x}) - \bar{x}\| < \epsilon, \quad \epsilon > 0,$$

where $\|.\|$ is a norm, ϵ is a well-chosen threshold. $\bar{x}$ is given as an arbitrary combination of local components

$$\bar{x} = \left(x_1^{k_1}, \ \ldots, \ x_p^{k_p}\right), \quad k_1, \ \ldots, \ k_p \in \mathbb{N}.$$

The major problem of termination detection is how to collect $x_i^{k_i}$ and execute the evaluations.

Recently, several developments for the asynchronous iterations have been proposed in different domains, such as domain decomposition methods [14, 15, 13], convergence detection methods [18, 11], and programming libraries [10, 12]. In this paper, we propose a modified recursive doubling algorithm applied to the non-blocking collective communication, which is addressed in the next section. In Section 3, we present some convergence detection strategies based on our new method. Finally, further discussion is given in Section 4 about the implementation and performance.

2 Modified Recursive Doubling

Recall that p is the number of processors. We define a $\mu_0 \in \mathbb{N}$ such that

$$p_0 = 2^{\mu_0} \leq p < 2^{\mu_0 + 1},$$

where p_0 denotes a pivot. Note that for the parallel iterative methods, the `Allreduce` function is very useful because we need to collect residual values from different processors. For the asynchronous iterations, however, the collective operations should be not only efficient, but also performed in a non-blocking way.

Traditional recursive doubling (see, e.g., [20]) is one of the possible algorithms for the `Allreduce` function. It involves a power-of-two number of processors, which adapts only to some special situations. We consider a modified version of recursive doubling, in which a backward shift and a forward shift are required in the general case. The first step is sending data from the extra processors to the first several processors, called backward shift, illustrated in Figure 1. During this process, we proceed the corresponding arithmetical operations, such as summation, maximization, and minimization. Then, the recursive doubling algorithm is proceeded only within the power-of-two processors to exchange data and execute reduction operation as shown in Figure 2. Finally, a forward shift is proceeded to send back the final data to the extra processors as illustrated in Figure 3, which is indeed the inverse operation of the first shift.

Asynchronous iterations require non-blocking communication, which can be implemented is several ways. For example, we might prefer to create a new thread for a desired collective

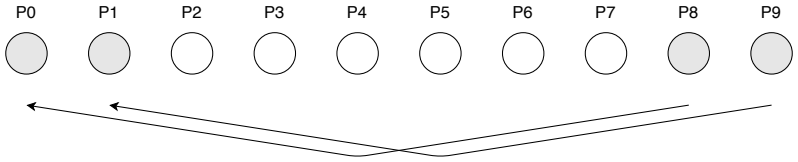


Figure 1: Backward shift

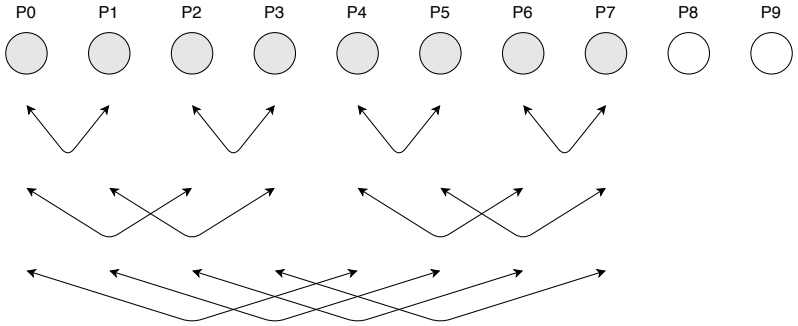


Figure 2: Recursive doubling

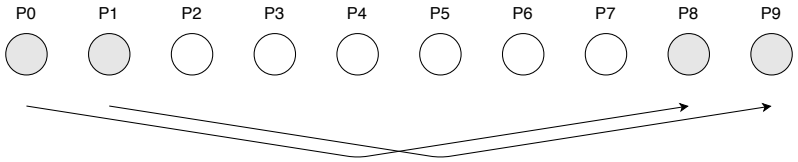


Figure 3: Forward shift

function, and then design its behavior by some external interface functions; we could also create a state-based interface that should be invoked repeatedly in user applications, in which some lightweight functions act as different states in the life cycle of a collective operation. Here, we adopt the latter and give an example of state diagram depicted in Figure 4, which implements a non-blocking `Allreduce` function. From the picture, we can see that each cycle begins with the backward shift operation. If the rank of processor belongs to the extra range, it sends data and enters into the forward shift state. In other cases, the relative processors must enter into the updating loop, which involves all the processors having a rank smaller than the pivot p_0. Finally, a forward shift process is executed to gather the final results to the non-power-of-two processors. Notice that the first several processors within the exponential area engage as well in the shift subroutines.

The amount of data exchanged by each processor depends on the way of collecting residual values. We need exactly $\log p_0 + 2$ steps to finish a cycle in the synchronous case. If there is only a floating point residual value being exchanged in each processor, then totally $p_0 \log p_0 + 2(p - p_0)$ data are exchanged in each cycle. In asynchronous mode, this number keeps the same. However, processors wait no longer the others and conduct iterations on their own pace.

3 Convergence Detection Algorithms

We could also develop other collective operations based on the backward-forward recursive doubling algorithm. In practice, however, these functions are rarely used in the context of asynchronous iterations because we expect to exploit the most recent values as much as

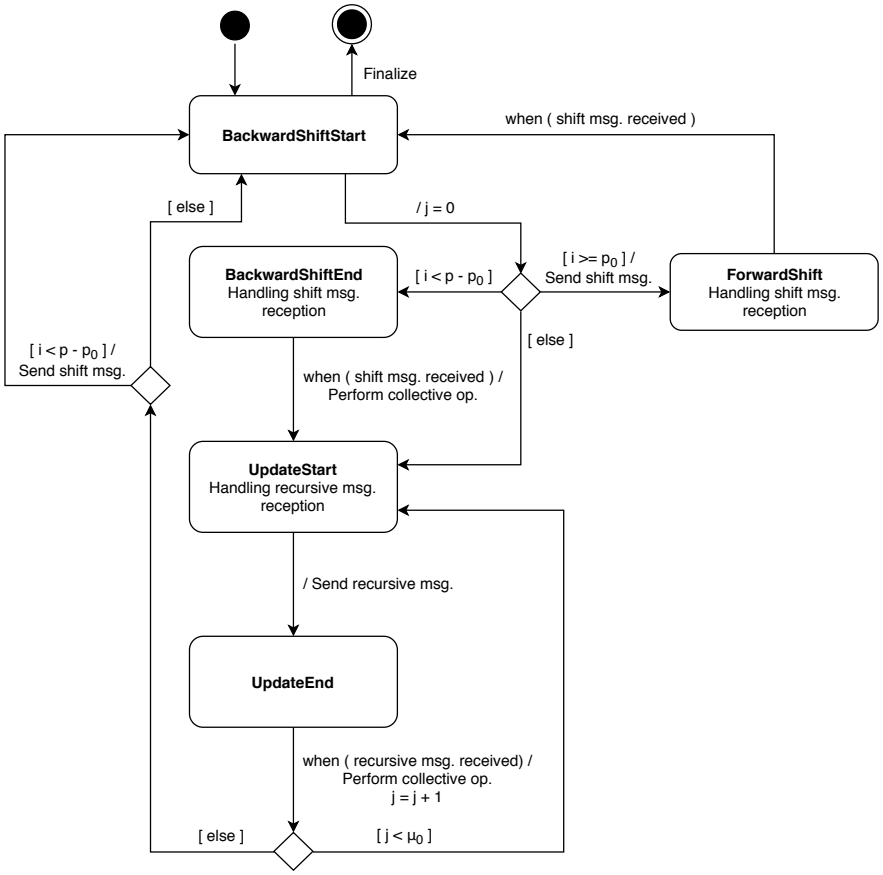


Figure 4: Statechart of a non-blocking `Allreduce` function

possible, which favors the point-to-point operations like `Send` and `Recv` functions. On the other hand, the residual collection requires intrinsically an `Allreduce` operation. Therefore, we address the convergence detection problem in terms of the non-blocking reduction.

We consider first an inexact residual collection strategy that involves only the `Allreduce` function, depicted as follows.

```
res_norm = res_thresh
res_loc = res_thresh
res_glb = res_thresh
flag = 1
while res_norm ≥ res_thresh do
  z_i = x_i
  Compute(x_i, A_i, b_i, x)
  Send(x_i)
  Allreduce(res_glb, res_loc, flag)
  if flag then
    res_norm = res_glb
    res_loc = ‖x_i − z_i‖_∞
    flag = 0
  end if
  Recv(x)
end while
```

We mention here that although such algorithm is not exact, it might be efficient due to the simplicity and still has an acceptable precision. In the algorithm, we take the maximum

norm as an example to compute residual and omit some function parameters, e.g., the arithmetic operation of reduction. Unlike the message-passing standard [16], our implementation is based on the state that requires the function invocation repeatedly, not just a request handler. The `Compute` function could be any appropriate iterative algorithm, such as Jacobi method or gradient method (see, e.g., [2]). This is inexact because `res_loc` might not be monotone all over the iterations. Sometimes global residual indicates a convergence signal but local residual rises instead due to the retard term.

Now we give a second version that leads to an exact solution in view of the residual collection, shown as follows.

```
res_norm = res_thresh
sflag = 1
eflag = 0
while res_norm ≥ res_thresh do
  Compute(x_i, A_i, b_i, x)
  Send(x_i)
  if sflag then
    Snapshot(x̄, x_i, eflag)
    if eflag then
      z_i = x̄_i
      Compute(x̄_i, A_i, b_i, x̄)
      res_loc = ‖x̄_i − z_i‖_∞
      sflag = 0
      eflag = 0
    end if
  else
    Allreduce(res_glb, res_loc, eflag)
    if eflag then
      res_norm = res_glb
      sflag = 1
      eflag = 0
    end if
  end if
  Recv(x)
end while
```

This algorithm involves a distributed snapshot process that generates a consistent solution buffer [3] (see also, e.g., [19, 11]). The snapshot algorithm first sends x_i to the processors that depend on x_i. In this situation, we call them dependent neighbors; then, processor i begins to wait for the necessary data from some other processors, which are called essential neighbors [2]; finally, it returns a collection of essential data that are used for the residual computation. Here we simplify the process by assuming that the communication follows an "all-to-all" pattern, which implies that both dependent neighbors and essential neighbors are all the other processors so that they are the same. For the general case, the algorithm would be similar. We first set `sflag` $= 1$ to enable snapshot process. Then, we compute `res_loc` when snapshot finishes and set all `sflag` $= 0$, which enables the reduction process. Finally, `Allreduce` is called repeatedly that is exactly the first algorithm, except that this time we keep a set of consistent data that provides an exact result.

4 Further Discussion

In this section we first discuss the implementation issue of the convergence detection algorithms. Here we take the message passing interface (MPI) standard as an example. Notice that in order to implement a non-blocking function, we should execute the relative instructions in an independent thread, which involves an explicit construction or an implicit

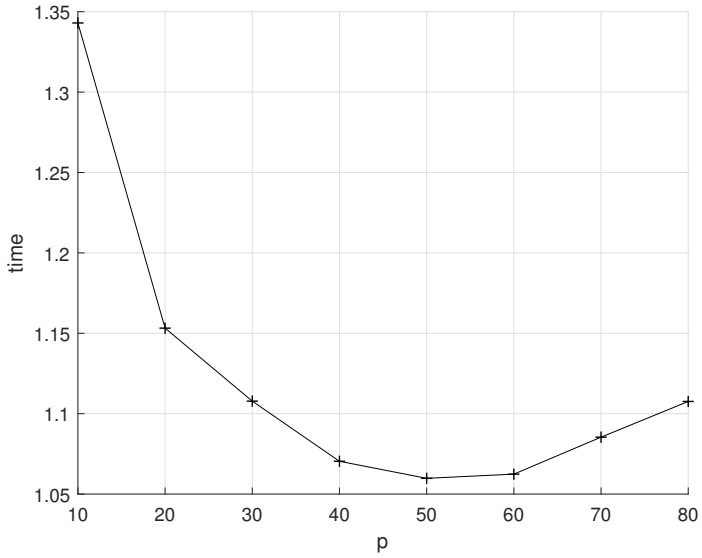


Figure 5: Asynchronous iterations in a concentrated environment

invocation. We choose the latter and invokes the non-blocking point-to-point instructions to exchange messages. We can use the external interface functions to generate a non-blocking function under the name of generalized requests. In current version, the two main functions are `MPI_Grequest_start` and `MPI_Grequest_complete`. In the next version, these functions will be redefined in order to provide a more flexible interface.

Notice that if the number of processors falls on the power-of-two case, the iterations in Figure 4 will jump over all the shift steps appropriately. Such case has been proven very efficient in several situations [20], whereas our algorithm can benefit from it as well. On the other hand, our `Send` operation is implemented in a blocking mode because it causes rarely a negative impact in practice on the efficiency. We could avoid wasting time by switching it to non-blocking mode without changing so much codes.

Finally, we mention here that our algorithm is suitable for a relatively "close" distributed environment; otherwise, there might be a great deal of communication operations exchanging data between long-distance nodes, which increases the transfer time. In such case, a tree-based algorithm is preferred. However, asynchronous iterations may not exhibit advantages in a completely local cluster, even perform sequentially like synchronous scheme with much more ongoing messages. Consider a two-point boundary value problem with an asynchronous relaxation solver [4]. We implement the mathematical operations by Alinea [6] and the asynchronous iterations by JACK [12], which have been proven very efficient for the large-scale scientific computing [7, 8, 9]. The finite difference scheme is adopted for the discretization. The matrix dimension $n = 10000$ and b is chosen arbitrarily from -10 to 10. Results are shown in Figure 5. We observe that the iteration curve shows synchronous behavior that exists a bottleneck within a specific range of processors. The experiment was performed on a cluster of Intel Xeon CPU E5-2670 v3, connected by FDR Infiniband network with 56 Gbit/s, which is concentrated and favors synchronous iterations. Furthermore, an "all-to-all" algorithm generates huge amounts of messages in the asynchronous mode, which makes the network too messy to be efficient. In this case, we prefer the traditional synchronous iterative scheme, even for large-scale parallel computing.

Acknowledgment

This work was supported by the French national programme LEFE/INSU and the project ADOM (Méthodes de décomposition de domaine asynchrones) of the French National Research Agency (ANR).

References

[1] G. M. Baudet. Asynchronous iterative methods for multiprocessors. *J. ACM*, 25(2):226–244, 1978.

[2] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[3] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[4] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2(2):199–222, 1969.

[5] M. N. El Tarazi. Some convergence results for asynchronous algorithms. *Numerische Mathematik*, 39(3):325–340, 1982.

[6] F. Magoulès and A.-K. Cheik Ahamed. Alinea: An advanced linear algebra library for massively parallel computations on graphics processing units. *The International Journal of High Performance Computing Applications*, 29(3):284–310, 2015.

[7] F. Magoulès, A.-K. Cheik Ahamed, and R. Putanowicz. Auto-tuned Krylov methods on cluster of graphics processing unit. *International Journal of Computer Mathematics*, 92(6):1222–1250, 2015.

[8] F. Magoulès, A.-K. Cheik Ahamed, and R. Putanowicz. Optimized Schwarz method without overlap for the gravitational potential equation on cluster of graphics processing unit. *International Journal of Computer Mathematics*, 93(6):955–980, 2015.

[9] F. Magoulès, A.-K. Cheik Ahamed, and A. Suzuki. Green computing on graphics processing units. *Concurrency and Computation: Practice and Experience*, 28(16):4305–4325, 2016.

[10] F. Magoulès and G. Gbikpi-Benissan. JACK: an asynchronous communication kernel library for iterative algorithms. *The Journal of Supercomputing*, 73(8):3468–3487, 2017.

[11] F. Magoulès and G. Gbikpi-Benissan. Distributed convergence detection based on global residual error under asynchronous iterations. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):819–829, 2018.

[12] F. Magoulès and G. Gbikpi-Benissan. JACK2: An MPI-based communication library with non-blocking synchronization for asynchronous iterations. *Advances in Engineering Software*, 119:116–133, 2018.

[13] F. Magoulès, G. Gbikpi-Benissan, and Q. Zou. Asynchronous iterations of Parareal algorithm for option pricing models. *Mathematics*, 6(4):1–18, 2018.

[14] F. Magoulès, D. B. Szyld, and C. Venet. Asynchronous optimized Schwarz methods with and without overlap. *Numerische Mathematik*, 137(1):199–227, 2017.

[15] F. Magoulès and C. Venet. Asynchronous iterative sub-structuring methods. *Mathematics and Computers in Simulation*, 145:34–49, 2018.

[16] Message Passing Interface Forum. MPI: A message passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4):159–416, 1994.

[17] J.-C. Miellou. Algorithmes de relaxation chaotique à retards. *ESAIM: Mathematical Modelling and Numerical Analysis*, 9(R1):55–82, 1975.

[18] J.-C. Miellou, P. Spiteri, and D. El Baz. A new stopping criterion for linear perturbed asynchronous iterations. *Journal of Computational and Applied Mathematics*, 219(2):471–483, 2008.

[19] S. A. Savari and D. P. Bertsekas. Finite termination of asynchronous iterative algorithms. *Parallel Computing*, 22(1):39–56, 1996.

[20] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.