Parallel Computing vs. Distributed Computing: A Great Confusion? (Position Paper)

Michel Raynal[1,2,3](✉)

[1] Institut Universitaire de France, Paris, France
[2] IRISA, Université de Rennes, Rennes, France
[3] Department of Computing, Hong Kong Polytechnic University, Hung Hom, Hong Kong
raynal@irisa.fr

"*Definierbar ist nur das, was keine Geschichte hat.*"
"*N'est définissable que ce qui n'a pas d'Histoire.*"
"*Only that which has no History can be defined.*"
—Friedrich Nietzsche (1844–1900).

"*Every sentence I utter must be understood not as an affirmation, but as a question.*"
—Niels Bohr (1885–1962).

Abstract. This short position paper discusses the fact that, from a *teaching point of view*, parallelism and distributed computing are often confused, while, when looking at their deep nature, they address distinct fundamental issues. Hence, appropriate curricula should be separately designed for each of them. The "everything is in everything (and reciprocally)" attitude does not seem to be a relevant approach to teach students the important concepts which characterize parallelism on the one side, and distributed computing on the other side.

1 A (Very) Quick Look at Parallel Computing

The main aim of parallelism is to produce efficient software. To that end, a lot of research on parallel systems lies at the frontier between programming languages, software engineering, scheduling algorithms, and technology. Moreover, advances in technology cannot be ignored. As noticed by M. Herlihy and V. Luchangco in [16], "Changes in technology can have far-reaching effects on theory. [...] After decades of being respected but not taken seriously, research on multiprocessor algorithms and data structures is going mainstream".

The class of problems which can be parallelized are basically sequential computing problems for which a static decomposition into a task graph, and appropriate scheduling strategies used at run-time, allow us to obtain efficient executions on specific target machines.[1] In a few words, parallelism aims at

[1] As explained later, designing a parallel algorithm to solve a given problem does not want to say that there is no sequential algorithm able to solve it (maybe very inefficiently).

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 41–53, 2015.
DOI: 10.1007/978-3-319-27308-2_4

producing (time-)efficient programs, and its fundamental issue consists in mastering *efficiency* [3,28][2].

2 What Is Distributed Computing

In a few words, *distributed computing* is about mastering *uncertainty*. Distributed computing arises when one has to solve a problem in terms of distributed entities (usually called processors, nodes, processes, actors, agents, sensors, peers, etc.) such that each entity has only a partial knowledge of the many parameters involved in the problem that has to be solved [32]. Hence, in one way or another, in any distributed computing problem, there are several computing entities, and each of them has to locally take a decision, whose scope is global. The uncertainty is not under the control of the programmer, it is created by the geographical scattering of the computing entities, the asynchrony of their communication, their mobility, the fact that each entity knows only a subset of the whole set of inputs (namely, its own local inputs), etc.

Although distributed algorithms are often made up of a few lines, their behavior can be difficult to understand and their properties hard to state, prove, and implement. Hence, distributed computing is not only a fundamental topic of *Informatics*[3], but also a challenging topic where simplicity, elegance, and beauty are first-class citizens [11,32].

The Notion of a (Distributed) Task. The basic unit of distributed computing is the notion of a *task*, which was formalized in several papers (e.g., see [18,19]). A task is made up of n processes p_1, ..., p_n (computing entities), such that each process has its own input (let in_i denote the input of p_i) and must compute its own output (let out_i denote the output of p_i). Let $I = [in_1, \cdots, in_n]$ be an input vector (let us notice that a process knows only its local input, it does not know the whole input vector). Let $O = [out_1, \cdots, out_n]$ be an output vector (similarly, even if a process is required to cooperate with the other processes, it will compute only its local output out_i, and not the whole output vector). A task T is defined by a set $\mathcal{I}$ of input vectors, a set $\mathcal{O}$ of output vectors, and a mapping T from $\mathcal{I}$ to $\mathcal{O}$, such that, given any input vector $I \in \mathcal{I}$, the output vector O (cooperatively computed by processes) is such that $O \in T(I)$. The case $n = 1$ corresponds to sequential computing (see Fig. 1). In this case a task boils down to a function.

[2] In a different domain, *real-time* computing is on mastering *on-time* computing [24].

[3] As nicely stated by E.W. Dijkstra (1920–2002): "*Computer science is no more about computers than astronomy is about telescopes*". Hence, to prevent ambiguities, I use the word *informatics* in place of *computer science*. On a pleasant side, there is no more "computer science" than "washing machine science".

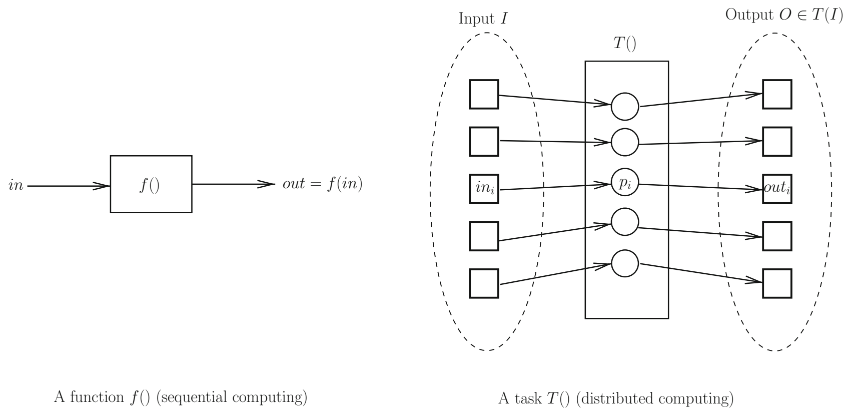
in
$f()$
$out = f(in)$
A function $f()$ (sequential computing)
Input I
$T()$
Output $O \in T(I)$
in_i
p_i
out_i
A task $T()$ (distributed computing)

Fig. 1. Function vs. task

3 A Fundamental Difference Between Parallel Computing and Distributed Computing

This difference lies in the fact that a task is distributed by its very definition. This means that the processes, each with its own inputs, are geographically distributed and, due to this imposed distribution, need to communicate to compute their outputs. The geographical *distribution of the computing entities is a not a design choice*, it is an input of the problem which gives its name to *distributed computing.*

Differently, in parallel computing, the inputs are, by essence, centralized. When considering the left part of Fig. 1, a function $f()$, and an input parameter x, parallel computing addresses concepts, methods, and strategies which allow to benefit from parallelism when one has to implement $f(x)$. The input x is given, and (if any) its initial scattering on distinct processors is not a priori imposed, but is a design choice aiming at obtaining efficient implementations of $f()$.

Hence, the *essence* of distributed computing is not on looking for efficiency but on coordination in the presence of "adversaries" such as asynchrony, failures, locality, mobility, heterogeneity, limited bandwidth, etc. From the local point of view of each computing entity, these adversaries create uncertainty generating non-determinism, which (when possible) has to be solved by an appropriate algorithm.

4 On the Computational Side: The *Hardness* of Distributed Computing

From a computability point of view, if the system is reliable, a distributed problem, abstracted as a task T, can be solved in a centralized way. Each process p_i sends its input in_i to a given predetermined process, which computes $T(I)$, and sends back to each process p_j its output out_j.

This is no longer possible if the presence of failures. Let us consider one of the less severe types of failures, namely process crash failures in an asynchronous system. One of the most fundamental impossibility result of distributed computing is the celebrated FLP result due to Fischer, Lynch, and Paterson [12]. This result states that it is impossible to design a deterministic algorithm solving the basic *consensus* problem in an asynchronous distributed system in which even a single process may crash, be the underlying communication medium a message-passing network of a read/write shared memory[4].

Hence, it appears that, in distributed computing, "there are many problems which are not computable, but these limits to computability reflect the difficulty of making decisions in the face of ambiguity, and have little to do with the inherent computational power of individual participants" [18]. As we can see, the essence of distributed computing is far from the efficiency issues motivating parallelism (See also [33]).

5 Parallel vs. Distributed Computing: A Schematic View

Inputs

– Parallel Computing: Inputs are "always" initially centralized. They can be "disseminated" as a design choice to benefit from parallelism. A problem is broken into distinct parts that can be solved concurrently. Parallelism was born from machines to overcome (some) inefficiency of sequential computing.
– Distributed Computing: Inputs are always distributed.

Main concepts

– Parallel Computing is looking for efficiency:
  Array processing, automatic parallelization, load balancing, machine architecture, scheduling [5,6], task graph [34], vector or systolic programming [10], pattern decomposition [27], etc.
– Distributed Computing is about mastering uncertainty:
  Local computation, non-determinism created by the environment, symmetry breaking, agreement, etc.

Outputs

– Parallel Computing: The outputs are a function of the inputs.
– Distributed Computing: The outputs are a function of both the inputs and (possibly) the environment[5].

[4] In the consensus problem each process is assumed to propose a value. The problem is defined by the three following properties. If a process do not crash, it has to decide a value (termination). No two processes decide differently (agreement). A decided value is a value that was proposed by a process (validity). As cooperating processes have to agree in one way or another (otherwise the problem is only a control flow problem), a lot of distributed computing problems rely on a solution to the consensus problem, or a variant of it.

[5] A simple example where the output depends on both the inputs and the environment is the *Non-Blocking Atomic Commit* problem (which is defined in Appendix A).

A few paradigmatic problems

- Parallel Computing: simulation, matrix computation, differential equations, etc.
- Distributed Computing: anonymous/oblivious agents (processes), local computing, rendezvous in arbitrary graphs, agreement problems, fault-tolerant cooperation, facing Byzantine failures.

We do not also have to forget that, in both cases (parallel computing or distributed computing), the underlying synchronization is a fundamental issue.

6 An Approach to Teach Distributed Computing

A proven and trusted way to teach the basics of sequential computing consists in teaching its basic components, i.e., *sequential algorithms*. As the algorithms are at the core of *informatics*, their knowledge, and the concepts and techniques they rely on, allow students to understand and master the fundamental concepts of sequential computing.

In a very similar way, I think that teaching basic distributed algorithms constitutes the best way for students to learn the basic elements that allow them to capture, understand, and master the specificity and main features of distributed computing, and discover that distributed computing cannot be reduced to a set processors enriched with a communication medium.

Books have been written on distributed algorithms and distributed computing, e.g., [2,9,20,22,26,35–37]. Differently from a collection of papers, each of these books presents its *view* of the domain, and its way to give a concrete expression to it. Hence, in the same spirit, I present in the following my *personal view* of teaching and introducing students to distributed computing, through distributed algorithms. This view is exposed more deeply in four books I wrote of the topic, each addressing a specific part of it. Of course, being personal, this view may be judged as both partial and questionable.

7 Distributed Algorithms at the Undergraduate Level

At the undergraduate level, I teach distributed algorithms in failure-free systems. The corresponding material is a subset of my book "*Distributed algorithms for message-passing systems*" [32], which is made up of seventeen chapters, structured in six parts:

- Distributed graph algorithms,
- Logical time and global state in distributed systems,
- Mutual exclusion and resource allocation,
- High level communication abstractions,
- Detection of properties of distributed computation,
- Distributed shared memory.

Distributed Graphs Algorithms. A simple way to introduce distributed algorithms to student consists in considering graph problems (such as vertex coloring, or maximal independent set). A processor is associated with each node and processors communicate by sending and receiving messages along channels defined by the edges of the graph. As each node (processor) has only a local view of the whole graph, these distributed graph algorithms lead students to think in distributed way, i.e., a processor can only act locally to solve a *global* problem involving all the nodes.

A Quick Look at Distributed Computability. As in sequential computing, students must be taught the limits of distributed computing from the very beginning. While the FLP result seems too much complicated for undergraduate students, I present in my introductory lectures the impossibility to elect a leader in an anonymous ring network [1]. For students, this is "similar" to the impossibility to design a sequential comparison-based algorithm that sorts an array of n elements in less than $O(n \log n)$ comparisons of elements.

The Nature of Distributed Computing. Then, I address the second part of the book and focus on the nature of distributed executions. This consists in two main presentations. The first one is on the fact that a distributed execution is a partial order on a set of events, and on Chandy-Lamport distributed snapshot algorithm. The second one is the introduction of logical time (mainly Lamport scalar time and vector time) and its use to solve distributed problems. Chandy-Lamport algorithm is particularly important as it allows students to grasp the deep nature of distributed computing: it determines a consistent global state of a distributed computation, but it is impossible to claim that the computation passed through this global state or not. This algorithm exhibits the *relativistic* nature of distributed computing (let us remind that distributed computing is on mastering uncertainty).

Other Topics Addressed in My Lectures. According to time, I then address mainly distributed mutual exclusion, construction of communication abstractions of higher level than send/receive communication (e.g., rendezvous, causal message delivery, total order message delivery), detection of properties of distributed computations, (such as deadlock detection and termination detection), and the construction of a distributed shared memory on top of a distributed asynchronous message-passing system.

I think that, due to their paradigmatic features, distributed algorithms solving the mutual exclusion are important to teach because they are based on distributed principles and techniques that can be used to solve many other problems (examples are given in [32]).

Differently from mutual exclusion, which is not a problem specific to a distributed setting, termination detection is. This problem consists in designing a distributed observation algorithm able to detect if an upper lying distributed application has or not terminated. This is a non-trivial problem[6]. Hence, termination detection introduces students to specific features imposed by distributed settings.

Finally, I use the construction of a distributed read/write shared memory on top of a message-passing system to introduce students to data consistency criteria in a distributed context.

8 Distributed Algorithms at the Graduate Level

At the graduate level (Master 2 and lectures for PhD students) I address distributed computing in the presence of failures. This makes problems much more difficult (or, sometimes, even impossible) to solve. To this end, I wrote three books, one devoted to the case where communication is through a shared memory, and two when it is through an underlying message-passing network (one considers asynchronous systems, while the other one considers synchronous systems). I examine below their contents and my associated lectures.

9 When Communication Is Through a Shared Memory

This book titled "*Concurrent programming: algorithm, principles, and foundations*" [31] is both on synchronization and distributed algorithms where processes communicate through a shared memory. It is composed of seventeen chapters, structured in six parts:

- Lock-based synchronization,
- On the foundations side: the atomicity concept,
- Mutex-free synchronization, and associated progress conditions (the most important being obstruction-freedom [17], lock-freedom –also called "non-blocking"– [21], and wait-freedom [15]),
- The transactional memory approach,
- On the foundations side: from safe bits to atomic registers,
- On the foundations side: the computability power of concurrent objects.

In my lectures, I mainly introduce and develop mutex-free synchronization, and the more theoretical part devoted to the computability power of concurrent objects.

Mutex-Free Synchronization. The design of concurrent objects is a fundamental issue, as those are the objects that allows processes to cooperate in the presence of concurrency. While lock-based algorithms are well-known to implement concurrent objects, they cannot cope with process failures (which can occur in a distributed setting such as a multicore). This is because, if a processor crashes

[6] Let us notice that, even if we could observe all processors simultaneously passive, we could not claim that the application terminated. This is because, messages can still be in transit, which will re-activate processors when they will arrive at their destination.

while holding a lock on an object, this object becomes and remains forever inaccessible. Let us remind that in an asynchronous system, it is impossible to distinguish a slow process from a crashed process (the distinction can be done in a synchronous system [30]). Hence, "modern" synchronization [15] (i.e., synchronization in asynchronous systems prone to process crash failures) cannot be solved by a simple patching of traditional synchronization mechanisms as described in [7,8].

Interestingly, (not all but) many concurrent objects have mutex-free implementations. Those are such that the algorithms implementing the object operations always terminate if the invoking process does not crash, where "always" means "despite the behavior of the other processes" (which can be slow or even crashed). These implementations are not trivial. This come from the fact that it is not possible to prevent several process from accessing simultaneously the internal representation of the concurrent object.

Hence, the aim of this part of the lectures is to make students aware of "modern" synchronization techniques (mainly wait-free computing), so that they are able to cope with the net effect of asynchrony and failures.

The Power of Concurrent Objects. Suppose you have to choose one of the following multicore machines. Both have the same number of processors but, in addition to atomic read/write registers, the hardware of machine A provides a $Test\&Set()$ operation, while the hardware of machine B provides a $Compare\&Swap()$ operation. Which machine do you choose?

In a failure-free context, and from a computability point of view, there is no difference. Actually in such a context, read/write atomic registers are sufficient to allow processes to cooperate. Differently, if processors may crash, machine B is much more powerful than machine A. This is related to the computability power of the operations $Test\&Set()$ and $Compare\&Swap()$, which is measured with the *consensus number* notion, introduced by Herlihy [15]. The consensus number of $Test\&Set()$ is 2, while the one of $Compare\&Swap()$ is $+\infty$. More precisely, this establishes a strict hierarchy on the synchronization power of concurrent objects. (For the interested reader, the consensus number notion is developed in Appendix B.)

The notions of a mutex-free implementation of concurrent objects, and consensus numbers, are fundamental concepts of modern synchronization (where, as aid previously, "modern" means "when we want to cope with the net effect of concurrency and any number of process failures"). They constitute the core of my lectures in systems where communication is through shared memory.

10 When Communication Is by Message-Passing

I addressed distributed computing in the presence of failures in two complementary books, which constitute the base of my teaching concerning failure-prone message-passing distributed systems. The first one is titled "*Communication and agreement abstractions for fault-tolerant asynchronous distributed systems*" [29], and second one is titled "*Fault-tolerant agreement in synchronous distributed systems*" [30].

The first one considers process crash failures in asynchronous systems. It composed of seven chapters, structured in three parts:

- The register abstraction,
- The uniform reliable broadcast abstraction,
- Agreement abstraction.

The second one considers synchronous systems, where processes may fail by crashing, committing omission failures, or behaving arbitrarily (Byzantine processes).

High Level Communication Abstraction. The first lectures are devoted to the construction of a reliable read/write shared memory on top of an asynchronous system prone to process failures. It is shown that this is impossible when half or more processes may crash. I then describe algorithms solving the problem when a majority of processes never crash. This allows students to understand the inherent cost of a shared memory built on top of a crash-prone asynchronous system, and consequently the cost of a strong cooperation object.

Synchrony vs. Asynchrony. A main point of the corresponding lectures is to obtain a deeper understanding of the net effect of asynchrony and failures. As an example, in the part of the lectures devoted to the consensus problem, I show that, while the problem can be solved in a synchronous system where any number of processes may crash, it is impossible to solve if the system is asynchronous, even if a single process may crash, and this is independent of the total number of processes (FLP impossibility). Hence, when considering the synchrony/asynchrony axis, a fundamental question for students is: which is the weakest synchrony assumptions that allows consensus to be solved. This is explored in details during several lectures.

11 Conclusion

While in the Middle Ages, philosophy, scholastic, and logic were considered as a single study domain, they are now considered as distinct domains. It is similar in mathematics, where (as a simple example and since a long time), algebra and calculus are considered as separate domains, each with its own objects, concepts, and tools. As another example, while solving an application may require knowledge in both probability, graph algorithms, and differential equations, those are considered as distinct mathematical areas.

It is the same in *informatics*, where some applications may involve at the same time parallelism and distributed computing. Life is diverse, and so are applications. But, from a teaching point of view, parallelism and distributed computing are distinct scientific areas, and their main concepts should consequently be taught separately to students. Teaching is not an accumulation of facts [23]. It is the exposure of concepts, principles, methodologies, and tools, that (among other targets) non only have to ensure that students will find a job when they finish their studies, but also (and maybe more importantly) ensure that, thanks to their methodological and conceptual background, students will still have a job in twenty-five years!

To conclude, let us remind that this is a position paper, where have been presented a few personal views. As nicely expressed by Niels Bohr:

"*Prediction is very difficult, especially when it is about the future.*"

A The Non-blocking Atomic Commit Problem

In some applications, each process executes some local computation, at the end of which, it votes *yes* or *no*, according to its local computation. Then, according to their votes, the processes have to collectively commit or abort their local computation. Named non-blocking atomic commit (NBAC), this problem is formalized as follows [4,13,14,29].

Let us consider n asynchronous processes prone to crash failures. Each process proposes a value *yes* or *no*, and has to decide the value COMMIT or the value ABORT. NBAC is defined by the following set of properties[7].

- Validity. A decided value is COMMIT or ABORT. Moreover,
  - Justification. If a process decides COMMIT, all the processes voted *yes*.
  - Obligation. If all processes voted *yes* and there is no crash, no process decides ABORT.
- Integrity. A process decides at most once.
- Agreement. No two processes decide different values.
- Termination. Each non-faulty process decides.

This problem has the same agreement, integrity and termination properties as the consensus problem. It differs from it in the validity property, namely, a decided value is not a proposed value but a value that depends on both the proposed values *and* the failure pattern. Differently, the properties defining the consensus problem do not refer directly to the failure pattern.

B Remark on the Notion of a Consensus Number of an Object

As previously indicated this notion was introduced by Herlihy [15]. It concerns the computability power of concurrent objects in shared memory systems where any number of processes may experience crash failures (e.g., multicore).

In such a context, a fundamental problem consists in building concurrent objects able to cope with any number of process failures. This is called the *wait-free* model. It is shown that the *consensus* object is *universal* in the sense

[7] This means that any algorithm solving the NBAC problem must satisfy all these properties.

that, given read/write registers and consensus objects, it is possible to design algorithms implementing (in the wait-free model) any concurrent object defined by a sequential specification. Such algorithms are called *universal constructions.*

Hence, the key for reliability in the wait-free model is the consensus object. This object can be informally defined as follows. Assuming each process proposes a value, the processes that do not crash have to decide the same value, and this value must be one of the proposed values. This apparently very simple object is impossible to implement in the basic wait-free model where processes communicate by accessing read/write registers only. This is the bad news [12,15,25][8].

But multiprocessor machines are usually endowed with specialized operations to address synchronization issues. Examples of such operations (or objects) are $Test\&Set()$, $Swap()$, $Fetch\&add()$, $Compare\&Swap()$, LL/SC, etc. Hence, the question: "Is the "bad news" still true in a multiprocessor enriched with such an operation?" Herlihy showed in [15] that it is possible to associate an integer with each of these synchronization operations, called *consensus number*, which characterizes its computability power in the wait-free model. More precisely, the consensus number of a synchronization operation is x if it allows to implement a consensus object in a system of x (or less) processes but not in a system of $(x+1)$ processes. It follows that consensus numbers provide us with a hierarchy measuring the computability power of hardware synchronization operations. The consensus number of read/write is 1. The consensus number of $Test\&Set()$ or $Swap()$ is 2; etc. until operations such as $Compare\&Swap()$ or LL/SC, whose consensus number is $+\infty$.

More developments on consensus objects, universal constructions, and Herlihy's hierarchy can be found in [15,31,33,36].

References

1. Angluin, D.: Local and global properties in networks of processors. In: Proceedings of the 12th ACM Symposium on Theory of Computation (STOC 1981), pp. 82–93, ACM Press (1981)
2. Attiya, H., Welch, J.L.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. Wiley-Interscience, 2nd edn, p. 414. Wiley, New York (2004)
3. Barney, B.: Introduction to parallel computing. https://computing.llnl.gov/tutorials/parallel_comp/#Whatis.UCRL-MI-133316
4. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems, p. 370. Addison Wesley, Reading (1987). ISBN 0-201-10715-5
5. Blazewicz, J., Pesch, E., Trystram, D., Zhang, G.: New perspectives in scheduling theory. J. Sched. **18**(4), 333–334 (2015)
6. Bouguerra, M.-M., Kondo, D., Mendonca, F.M., Trystram, D.: Fault-tolerant scheduling on parallel systems with non-memoryless failure distributions. J. Parallel Distrib. Comput. **74**(5), 2411–2422 (2014)

[8] Let us recall that, from a computability point of view, read and write operations are sufficient to solve any problem (in a Turing's sense, i.e., computable by Turing's machine) when there are no failures. The "bad news" says that this is no longer true when there are failures.

7. Brinch Hansen, P.: The Architecture of Concurrent Programs, p. 317. Prentice Hall, Englewood Cliffs (1977)
8. Brinch Hansen, P. (ed.): The Origin of Concurrent Programming, p. 534. Springer, New York (2002)
9. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to Reliable and Secure Distributed Programming, p. 367. Springer, Heidelberg (2012). ISBN 978-3-642-15259-7
10. Chandy, K.M., Misra, J.: Parallel Program Design, p. 516. Addison Wesley, Reading (1988)
11. Dijkstra, E.W.: Some beautiful arguments using mathematical induction. Algorithmica **13**(1), 1–8 (1980)
12. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
13. Gray, J.: Notes on Database Operating Systems: and Advanced Course. LNCS, vol. 60, pp. 10–17. Springer, Heildelberg (1978)
14. Hadzilacos, V.: On the relationship between the atomic commitment and consensus problems. In: Simons, B., Spector, A. (eds.) Fault-Tolerant Distributed Computing. LNCS, vol. 448, pp. 201–208. Springer, Heildelberg (1990)
15. Herlihy, M.P.: Wait-free synchronization. Trans. Program. Lang. Syst. **13**(1), 124–149 (1991)
16. Herlihy, M.P., Luchangco, V.: Distributed computing and the multicore revolution. ACM SIGACT News **39**(1), 62–72 (2008)
17. Herlihy, M.P., Luchangco, V., Moir, M.: Obstruction-free synchronization: double-ended queues as an example. In: Proceedings of 23th International IEEE Conference on Distributed Computing Systems (ICDCS 2003), pp. 522–529. IEEE Press (2003)
18. Herlihy, M.P., Rajsbaum, S., Raynal, M.: Power and limits of distributed computing shared memory models. Theor. Comput. Sci. **509**, 3–24 (2013)
19. Herlihy, M., Shavit, N.: The topological structure of asynchronous computability. J. ACM **46**(6), 858–923 (1999)
20. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, p. 508. Morgan Kaufmann, Burlington (2008). ISBN 978-0-12-370591-4
21. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
22. Kshemkalyani, A.D., Singhal, M.: Distributed Computing: Principles, p. 736. Algorithms and Systems. Cambridge University Press, Cambridge (2008)
23. Lamport, L.: Teaching concurrency. ACM SIGACT News, Distrib. Comput. Column **40**(1), 58–62 (2009)
24. Liu, C.L.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM **20**(1), 46–61 (1973)
25. Loui, M., Abu-Amara, H.: Memory requirements for agreement among unreliable asynchronous processes. In: Advances in Computing Research, vol. 4, pp. 163–183. JAI Press (1987)
26. Lynch, N.A.: Distributed Algorithms, p. 872. Morgan Kaufmann, San Francisco (1996)
27. Matton, T., Sanders, B.A., Massingill, B.: Patterns for Parallel Programming, p. 384. Addison Wesley, Reading (2004). ISBN 978-0-321-22811-6
28. Padua, D. (ed.): Encyclopedia of Parallel Computing, p. 2180. Springer, Heildelberg (2011). ISBN 978-0-387-09765-7

29. Raynal, M.: Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems, p. 251. Morgan & Claypool Pub. (2010). ISBN 978-1-60845-293-4
30. Raynal, M.: Fault-Tolerant Agreement in Synchronous Distributed Systems, p. 167. Morgan & Claypool (2010). ISBN 978-1-608-45525-6
31. Raynal, M.: Concurrent Programming: Algorithms, Principles, and Foundations, p. 530. Springer, Heildelberg (2013). ISBN 978-3-642-32026-2
32. Raynal, M.: Distributed Algorithms for Message-passing Systems, p. 515. Springer, Heilderberg (2013). ISBN 978-3-642-38122-5
33. Raynal, M.: What can be computed in a distributed system? In: Bensalem, S., Lakhneck, Y., Legay, A. (eds.) From Programs to Systems. LNCS, vol. 8415, pp. 209–224. Springer, Heidelberg (2014)
34. Robert, Y.: Task graph scheduling. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 2013–2025. Springer, Heilderberg (2011)
35. Santoro, N.: Design and Analysis of Distributed Algorithms, p. 589. Wiley, New York (2007)
36. Taubenfeld, G.: Synchronization Algorithms and Concurrent Programming, p. 423. Upper Saddle River, Pearson Education/Prentice Hall (2006). ISBN 0-131-97259-6
37. Tel, G.: Introduction to Distributed Algorithms, p. 596. Cambridge University Press, Cambridge (2000). ISBN 0-521-79483-8