Toward Structured Parallel Programming: Send-Receive Considered Harmful

Sergei Gorlatch(✉)

University of Muenster, Münster, Germany
gorlatch@uni-muenster.de

Abstract. During the software crisis of the 1960s, Dijkstra's famous thesis *"goto considered harmful"* paved the way for structured programming. In this paper that is a modified version of the short communication [10], we suggest that many current difficulties of parallel programming based on message passing are caused by poorly structured communication, which is a consequence of using low-level *send-receive* primitives. We argue that, like *goto* in sequential programs, *send-receive* should be avoided as far as possible and replaced by *collective operations* in the setting of message passing. We dispute some widely held opinions about the apparent superiority of low-level, pairwise primitives over collective operations, and we present substantial theoretical and empirical evidence to the contrary in the context of MPI (Message Passing Interface).

We also briefly mention our recent results obtained in the broader context of programming for modern many-core parallel systems.

Keywords: Programming methodology · Parallel systems · Structured programming · Message Passing Interface (MPI)

1 Introduction

The development of software for modern parallel and distributed systems is still a challenging and difficult task. One of the obvious reasons for this unsatisfactory situation is that today's programmers rely mostly on the programming culture of the 1980s and '90s, the Message Passing Interface (MPI) [15] still being the programming tool of choice for demanding applications.

The main advantage of MPI is that in the 1980s it integrated and standardized parallel constructs that were proven in practice. This put an end to the unacceptable previous situation when every hardware vendor provided its own set of communication primitives, and those primitives sometimes differed even between different brands of the same machine.

In order to enable high performance, MPI's communication management based on low-level primitives *send* and *receive* results in a complicated programming process. Several attempts were made to overcome this (e.g. HPF and OpenMP). However, despite reported success stories, these approaches have

T. Margaria et al. (Eds.): Steffen Festschrift, LNCS 11200, pp. 208–217, 2019.
https://doi.org/10.1007/978-3-030-22348-9_13

never achieved the popularity of MPI, mostly because they make the performance of parallel programs less understandable and difficult to predict.

A similar "software crisis" occurred in the sequential setting in the 1960s. The breakthrough was made by Dijkstra in his famous letter "*goto* considered harmful" [5], in which the finger of blame was pointed at the *goto* statement. By that time, [3] had formally demonstrated that programs could be written without any *goto* statements, in terms of only three control structures – sequence, selection and repetition. The notion of so-called *structured programming* [4] became almost synonymous with "goto elimination".

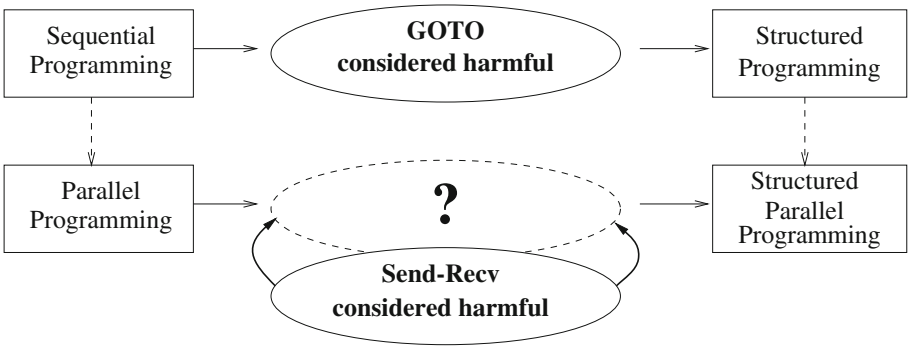

Fig. 1. As *goto* in the sequential case, *send-receive* complicates parallel programming.

Bernhard Steffen et al. [21] demonstrated that structured, rigorous programming greatly improves the formal analyses of important properties of parallel programs. In order to benefit from the experience in structured programming, we should answer the question: Which concept/construct plays a similar harmful role to that of *goto* in the parallel setting? As shown in Fig. 1 and demonstrated from Sect. 2 onwards, we suggest *send-receive* statements to be "considered harmful" and avoided as far as possible in parallel MPI programs.

The thrust of this paper is:
Parallel programming based on message passing can be improved by expressing communication in a structured manner, without using send-receive statements.

We demonstrate the advantages of collective operations over *send-receive* in five areas: simplicity, expressiveness, programmability, performance and predictability. This paper is a slightly modified version of [10]. The structured approach has been recently extended in different areas of parallel programming. In particular, novel parallel architectures like multi-core CPUs and many-core GPUs (Graphics Processing Units) require structured parallel programming at the node level, as an alternative to the low-level CUDA and OpenCL approaches, while message passing considered in this paper remains relevant for parallelizing across nodes. For our recent results, we refer the reader to the survey on algorithmic skeletons [11], the SkelCL library [26], skeleton-based transformations [16], and the LIFT approach [17].

2 The Challenge of Simplicity

Myth: Send-receive primitives are a simple way of specifying communication in parallel programs.

To reason effectively about a parallel program comprising hundreds or thousands of processes, one needs a suitable abstraction level. Programmers usually think in terms of how data has to be distributed to allow local computation: there is a stage (phase) of computation followed by a stage of communication, these stages being either synchronized, as in the BSP model [28], or not. Collectives neatly describe data redistributions between two stages, while individual sends and receives do not match this natural view, which leads to the following problems:

- There is no simple set of coordinates that describe the progress of a parallel program with individual communication. Such programs are therefore hard to understand and debug.
- If MPI is our language of choice, then we have not just one *send-receive*, but rather eight different kinds of *send* and two different kinds of *receive*. Thus, the programmer has to choose among 16 combinations of *send-receive*, some of them with very different semantics (blocking/non-blocking, synchronous/asynchronous, buffered/non-buffered, etc). Of course, this makes message-passing programming very flexible, but even less comprehensible!
- The last but not least problem is the size of programs. For example, a program for data broadcasting using `MPI_Bcast` may have only three instead of its *send-receive* equivalent's 31 lines of code [9,23].

Reality: The apparent simplicity of *send-receive* turns out to be the cause of large program size and complicated communication structure, which make both the design and debugging of parallel programs difficult.

3 The Challenge of Programmability

Myth: The design of parallel programs is so complicated that it will probably always remain an *ad hoc* activity rather than a systematic process.

The structure of programs with collective operations (a.k.a. collectives) as a sequence of stages facilitates high-level program transformations. A possible kind of transformation fuses two consecutive collective operations into one.

This is illustrated in Fig. 2 for a program with `p` processes, where each process either follows its own control flow, depicted by a down-arrow, or participates in a collective operation, depicted by a shaded area. Fusing two collective operations into one may imply a considerable saving in execution time; more on that in Sect. 6.

A particular fusion rule (1) states that, if operators `op1` and `op2` are associative and `op1` distributes over `op2`, then the following transformation of a composition of scan and reduction is applicable.

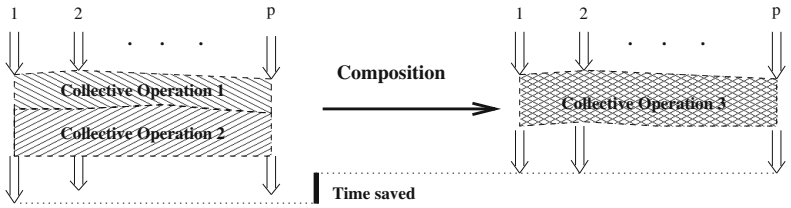

Fig. 2. The idea of fusing collective operations by a transformation like (1).

Here, function `Make_pair` duplicates its arguments, thus creating a pair, and `Take_first` yields the first component of a pair. Both functions are executed without interprocessor communication. The binary operator `f(op1,op2)` on the right-hand side works on pairs of values and is built using the operators from the left-hand side of the transformation. The precise definition of `f`, as well as other similar transformations, can be found in [8].

$$\left[\begin{array}{l}\texttt{MPI\_Scan (op1);}\\ \texttt{MPI\_Reduce (op2);}\end{array}\right] \Longrightarrow \left[\begin{array}{l}\texttt{Make\_pair;}\\ \texttt{MPI\_Reduce (f(op1,op2));}\\ \texttt{if my\_pid==ROOT then Take\_first;}\end{array}\right] \tag{1}$$

Rule (1) and other rules from [8] have the advantage that they are (a) proved formally as theorems, (b) parameterized by the occurring operators, e.g. `op1` and `op2`, and therefore customizable for a particular application, (c) valid for all possible implementations of collective operations, and (d) applicable independently of the parallel target architecture, and (e) suitable for automation.

Besides fusion rules, there are also transformations that decompose one collective operation into a sequence of smaller operations. Composition and decomposition rules can sometimes be applied in sequence, thus leading to more complex transformations, for example:

$$\left[\begin{array}{l}\texttt{MPI\_Scan(op1);}\\ \texttt{MPI\_Allreduce(op2);}\end{array}\right] \Longrightarrow \left[\begin{array}{l}\texttt{Make\_pair;}\\ \texttt{MPI\_Reduce-scatter(f(op1,op2));}\\ \texttt{Take\_first;}\\ \texttt{MPI\_Allgather;}\end{array}\right]$$

Profound results have been achieved with formalisms for the verification of concurrent and message-passing programs (see [25] for a very good overview of the state of the art). With collective operations, we take a different approach: we design message-passing programs in a stepwise manner (see [8]) by applying semantically sound transformations like (1). In Sect. 6, we show that such design process can be geared to predicting and improving performance.

Reality: Collective operations facilitate high-level program transformations that can be applied in a systematic program-design process.

4 The Challenge of Expressiveness

Myth: Collective operations are too inflexible and, therefore, unable to express many important applications.

To refute this quite widely held opinion, we present in Table 1 several important applications, which according to the recent literature were implemented using collective operations only, without notable performance loss compared with their counterparts using *send-receive*.

Table 1. Applications expressed using collective operations only

Application	Communication/Computation Pattern
Polynomial Multiplication	`Bcast (group); Map; Reduce; Shift`
Polynomial Evaluation	`Bcast; Scan; Map; Reduce`
Fast Fourier Transform	`Iter ( Map; All-to-all (group))`
Molecular Simulation	`Iter (Scatter; Reduce; Gather)`
N-Body Simulation	`Iter ( All-to-all; Map)`
Matrix Multiplication (SUMMA)	`Scatter; Iter (Scatter; Bcast; Map); Gather`
Matrix Multiplication (3D)	`Allgather (group); Map; All-to-all; Map`

Here, `Map` stands for local computations performed in the processes without communication; `Shift` is a cyclic, unidirectional exchange between all processes; `Iter` denotes repetitive action; `(group)` means that the collective operation is applied not to all processes of the program, but rather to an identified subset of processes (in MPI, it can be specified by a communicator).

Additional confirmation of the expressive power of collective operations is provided by the PLAPACK package for linear algebra [7], which has been implemented entirely without individual communication primitives.

Moreover, in one of the best textbooks on parallel algorithms [22], the whole methodology centres on implementing and then composing collective operations.

In paper [6], we proved the Turing universality of a programming language based on just two recursive collective patterns – anamorphisms and catamorphisms. This fact can be viewed as a counterpart to the "structured program theorem" by Böhm and Jacopini [3] for parallel programming.

Reality: A broad class of communication patterns found in important parallel applications is covered by collective operations.

5 The Challenge of Performance

Myth: Programs using *send-receive* are, naturally, faster than their counterparts using collective operations.

The usual performance argument in favour of individual communication is that collective operations are themselves implemented in terms of individual

send-receive and thus cannot be more efficient than the latter. However, there are two important aspects here that are often overlooked:

1. The implementations of collective operations are written by the implementers, who are much more familiar with the parallel machine and its network than an application programmer can be. Recent algorithms for collective communication [24] take into account specific characteristics of the interprocessor network, which can then be considered during the compilation phase of the communication library. The MagPIe library is geared to wide-area networks of clusters [20]. In [27], the tuning for a given system is achieved by conducting a series of experiments on the system. When using *send-receive*, the communication structure would probably have to be re-implemented for every new kind of network.
2. Very often, collective operations are implemented not via *send-receive*, but rather directly in the hardware, which is simply impossible at the user level. This allows all machine resources to be fully exploited and sometimes leads to rather unexpected results: e.g. a simple bidirectional exchange of data between two processors using *send-receive* on a Cray T3E takes twice as long as a version with two broadcasts [1]. The explanation for this phenomenon is that the broadcast is implemented directly on top of the shared-memory support of the Cray T3E.

Below, we dispute some other commonly held opinions about the performance superiority of *send-receive*, basing our arguments on empirical evidence from recent publications.

- It is not true that non-blocking versions of *send-receive*, `MPI_Isend` and `MPI_Irecv`, are invariably fast owing to the overlap of communication with computation. As demonstrated by [1], these primitives often lead to slower execution than the blocking version because of the extra synchronization.
- It is not true that the flexibility of *send-receive* allows faster algorithms than the collective paradigm. Research has shown that many designs using *send-receive* eventually lead to the same high-level algorithms as obtained by the "batch" approach [14]. In fact, batch versions often run faster [18].
- It is not true that the routing of individual messages over a network offers fundamental performance gains as compared with the routing for collective operations. As shown formally in [28], the performance gap in this case becomes, with large probability, arbitrarily small for large problem sizes.

Reality: There is strong evidence that *send-receive* does not offer fundamental performance advantages over collective operations. The latter offer machine-tuned, efficient implementations without changing the applications themselves.

6 The Challenge of Predictability

Myth: Reliable performance data for parallel programs can only be obtained *a posteriori*, i.e. by actually running the program on a particular machine configuration.

Performance predictability is, indeed, often even more difficult to achieve than absolute performance itself. Using collective operations, not only can we design programs by means of the transformations presented in Sect. 3; we can also estimate the impact of every single transformation on the program's performance. Table 2 contains a list of transformations from [12], together with the conditions under which these transformations improve performance.

Table 2. Impact of transformations on performance

Composition rule	Improvement if
`Scan_1; Reduce_2` $\rightarrow$ `Reduce`	always
`Scan; Reduce` $\rightarrow$ `Reduce`	$t_s > m$
`Scan_1; Scan_2` $\rightarrow$ `Scan`	$t_s > 2m$
`Scan; Scan` $\rightarrow$ `Scan`	$t_s > m(t_w + 4)$
`Bcast; Scan` $\rightarrow$ `Comcast`	always
`Bcast; Scan_1; Scan_2` $\rightarrow$ `Comcast`	$t_s > m/2$
`Bcast; Scan; Scan` $\rightarrow$ `Comcast`	$t_s > m(\frac{1}{2}t_w + 4)$
`Bcast; Reduce` $\rightarrow$ `Local`	always
`Bcast; Scan_1; Reduce_2` $\rightarrow$ `Local`	always
`Bcast; Scan; Reduce` $\rightarrow$ `Local`	$t_w + \frac{1}{m} \cdot t_s \geq \frac{1}{3}$

In the above table, a binomial-tree implementation of collective operations is presumed, our cost model having the following parameters: start-up/latency t_s, transfer time t_w and block size m, with the time of one computation operation assumed as the unit. These parameters are used in the conditions listed in the right column of the table. The estimates were validated in experiments on a Cray T3E and a Parsytec GCel 64 (see [8] for details).

Since the performance impact of a particular transformation depends on the parameters of both the application and the machine, there are alternatives to choose from in a particular design. Usually, the design process can be captured as a tree, one example of which is given in Fig. 3.

The best design decision is obtained by checking the design conditions, which depend either on the problem properties, e.g. the distributivity of operators, or on the characteristics of the target machine (number of processors, latency and bandwidth, etc.). For example, if the distributivity condition holds, it takes us from the root into the left subtree in Fig. 3. If the block size in an application is small, `Condition 1` (defined in [8]) yields "no", and we thus end up with the second (from left to right) design alternative, where `op3`= f(op1,op2) according to rule (1). Note that the conditions in the tree of alternatives may change for a different implementation of the collective operations involved.

Arguably, *send-receive* allows a more accurate performance model than collective operations do. Examples of well-suited models for finding efficient implementations are LogP and LogGP [19]. However, these models are overly detailed

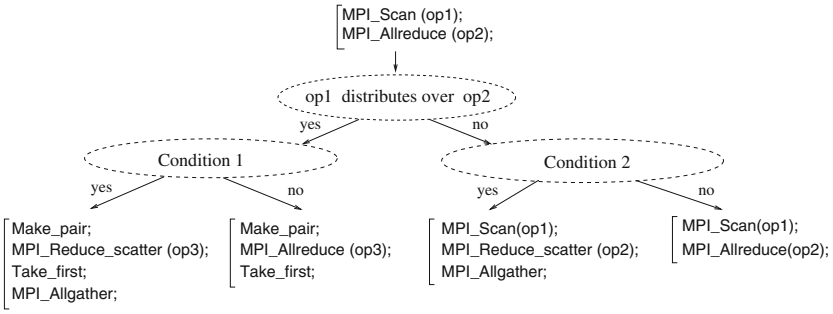

Fig. 3. The tree of design alternatives with decisions made in the nodes.

and difficult for an application programmer to use, as demonstrated by a comparison with batch-oriented models [2,13].

Reality: Collective operations contribute to the challenging goal of predicting program characteristics during the design process, i.e. without actually running the program on a machine. The use of *send-receive* obviously makes the program's performance much less predictable. Furthermore, the predictablity of collective operations greatly simplifies the modelling task at the application level, as compared with models like LogP.

7 Conclusion

This short communication proposes – perhaps somewhat polemically – viewing the *send-receive* primitives as harmful and, consequently, trying to avoid them in parallel programming.

We demonstrate the advantages of collective operations over *send-receive* in five major areas, which we call challenges: simplicity, expressiveness, programmability, performance and predictability. Based on recent publications in the field and our own research, we present hard evidence that many widely held opinions about *send-receive* vs. collective operations are mere myths.

Despite the success of structured programming, *goto* has not gone away altogether, but has either been hidden at lower levels of system software or packaged into safe language constructs. Similarly, there are parallel applications where non-determinism and low-level communication are useful, e.g. a taskqueue-based search. This motivates the development of "collective design patterns" or skeletons which should provide more complex combinations of both control and communication than the currently available collective operations of MPI.

We conclude by paraphrasing Dijkstra's famous letter [5], which originally inspired our work. Applied to message passing, it might read as follows:

> The various kinds and modes of send-receive used in the MPI standard, *buffered, synchronous, ready, (non-)blocking*, etc., are just too primitive; they are too much an invitation to make a mess of one's parallel program.

It is our strong belief that higher-level patterns, in particular collective operations, have good potential for overcoming this problem and enabling the design of well-structured, efficient parallel programs based on message passing.

Acknowledgements. I am grateful to many colleagues in the field of parallel computing, whose research provided necessary theoretical and experimental evidence to support the ideas presented here. It is my pleasure to acknowledge the very helpful comments of Chris Lengauer, Robert van de Geijn, Murray Cole, Jan Prins, Thilo Kielmann, Holger Bischof, and Phil Bacon on the preliminary version of the manuscript. The anonymous referees of [10] did a great job in improving the presentation.

References

1. Bernaschi, M., Iannello, G., Lauria, M.: Experimental results about MPI collective communication operations. In: Sloot, P., Bubak, M., Hoekstra, A., Hertzberger, B. (eds.) HPCN-Europe 1999. LNCS, vol. 1593, pp. 774–783. Springer, Heidelberg (1999). https://doi.org/10.1007/BFb0100638
2. Bilardi, G., Herley, K., Pietracaprina, A., Pucci, G., Spirakis, P.: BSP vs. LogP. In: Eighth ACM Symposium on Parallel Algorithms and Architectures, pp. 25–32 (1996)
3. Böhm, C., Jacopini, G.: Flow diagrams, Turing machines and languages with only two formation rules. Commun. ACM **9**, 366–371 (1966)
4. Dahl, O.J., Dijkstra, E.W., Hoare, C.A.: Structured Programming. Academic Press, London (1975)
5. Dijkstra, E.W.: Go To statement considered harmful. Commun. ACM **11**(3), 147–148 (1968)
6. Fischer, J., Gorlatch, S.: Turing universality of morphisms for parallel programming. In: Gorlatch, S., Lengauer, C. (eds.) Third Int. Workshop on Constructive Methods for Parallel Programming (CMPP 2002). Forschungsberichte der Fakultät IV - Elektrotechnik und Informatik, vol. 2002/07, pp. 81–98. Technische Universität Berlin, June 2002
7. van de Geijn, R.: Using PLAPACK: Parallel Linear Algebra Package. Scientific and Engineering Computation Series. MIT Press, Cambridge (1997)
8. Gorlatch, S.: Towards formally-based design of message passing programs. IEEE Trans. Softw. Eng. **26**(3), 276–288 (2000). http://wwwmath.uni-muenster.de/pvs/publikationen/papers/GorTSE.ps.gz
9. Gorlatch, S.: Send-recv considered harmful? myths and truths about parallel programming. In: Malyshkin, V. (ed.) PaCT 2001. LNCS, vol. 2127, pp. 243–257. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44743-1_24
10. Gorlatch, S.: Send-receive considered harmful: myths and realities of message passing. ACM TOPLAS **26**(1), 47–56 (2004)
11. Gorlatch, S.: Parallel skeletons. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 1417–1422. Springer, Boston, MA (2011). https://doi.org/10.1007/978-0-387-09766-4_24
12. Gorlatch, S., Wedler, C., Lengauer, C.: Optimization rules for programming with collective operations. In: Atallah, M. (ed.) Proceeding of the IPPS/SPDP 1999, pp. 492–499. IEEE Computer Society Press (1999)
13. Goudreau, M.W., Lang, K., Rao, S.B., Suel, T., Tsantilas, T.: Towards efficiency and portablility. Programming with the BSP model. In: Eighth ACM Symposium on Parallel Algorithms and Architectures, pp. 1–12 (1996)

14. Goudreau, M., Rao, S.: Single-message vs. batch communication. In: Heath, M., Ranade, A., Schreiber, R. (eds.) Algorithms for Parallel Processing, pp. 61–74. Springer, New York (1999)
15. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programmingwith the Message Passing. MIT Press, Cambridge (1994)
16. Hagedorn, B., Steuwer, M., Gorlatch, S.: A transformation-based approach to developing high-performance GPU programs. In: Petrenko, A.K., Voronkov, A. (eds.) PSI 2017. LNCS, vol. 10742, pp. 179–195. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74313-4_14
17. Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., Dubach, C.: High performance stencil code generation with Lift. In: Proceedings ACM CGO 2018, pp. 100–112 (2018). Best paper award
18. Hwang, K., Xu, Z.: Scalable Parallel Computing. McGraw Hill, New York (1998)
19. Kielmann, T., Bal, H.E., Gorlatch, S.: Bandwidth-efficient collective communication for clustered wide area systems. In: Parallel and Distributed Processing Symposium (IPDPS 2000), pp. 492–499 (2000)
20. Kielmann, T., Hofman, R.F., Bal, H.E., Plaat, A., Bhoedjang., R.A.: MagPIe: MPI's collective communication operations for clustered wide area systems. In: Proceedings of the ACM PPoPP, pp. 131–140 (1999)
21. Knoop, J., Steffen, B., Vollmer, J.: Parallelism for free: efficient and optimal bitvector analyses for parallel programs. ACM TOPLAS **18**(3), 268–299 (1996)
22. Kumar, V., et al.: Introduction to Parallel Computing. Benjamin/Cummings Publ, Redwood City (1994)
23. Pacheco, P.: Parallel Programming with MPI. Morgan Kaufmann Publ, San Francisco (1997)
24. Park, J.Y.L., Choi, H.A., Nupairoj, N., Ni, L.M.: Construction of optimal multicast trees based on the parameterized communication model. In: Proceedings of the International Conference on Parallel Processing (ICPP), vol. I, pp. 180–187 (1996)
25. Schneider, F.B.: On Concurrent Programming. Springer-Verlag, New York (1997). https://doi.org/10.1007/978-1-4612-1830-2
26. Steuwer, M., Gorlatch, S.: Skelcl: A high-level extension of OpenCL formulti-GPU systems. J. Supercomput. **69**(1), 25–33 (2014). https://doi.org/10.1007/s11227-014-1213-y
27. Vadhiyar, S.S., Fagg, G.E., Dongarra, J.: Automatically tuned collective communications. In: Proceedings of the Supercomputing 2000. Dallas, TX, November 2000
28. Valiant, L.G.: General purpose parallel architectures. In: Handbook of Theoretical Computer Science, vol. A, Chap. 18, pp. 943–971. MIT Press (1990)