Send-Receive Considered Harmful: Myths and Realities of Message Passing

SERGEI GORLATCH
Universität Münster

During the software crisis of the 1960s, Dijkstra's famous thesis *"goto considered harmful"* paved the way for structured programming. This short communication suggests 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 pairwise communication over collective communication and present substantial theoretical and empirical evidence to the contrary in the context of MPI (Message Passing Interface).

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms: Algorithms, Languages, Performance, Experimentation, Measurement

Additional Key Words and Phrases: Programming methodology, Message Passing Interface (MPI)

1. INTRODUCTION

Program development for parallel and distributed systems remains 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) [Gropp et al. 1994] still being the programming tool of choice for demanding applications.

The merit of MPI is that it integrated and standardized parallel constructs that were proven in practice. Designed to enable high performance, MPI's low-level communication management using the primitives *send* and *receive* results in a complicated programming process. Several attempts have been made to overcome this (e.g., HPF and OpenMP). However, despite reported success stories, these approaches have 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

Author's address: Universität Münster, Institut für Informatik, Einstrasse 62, D-48149 Münster, Germany; email: gorlatch@math.uni-muenster.de.

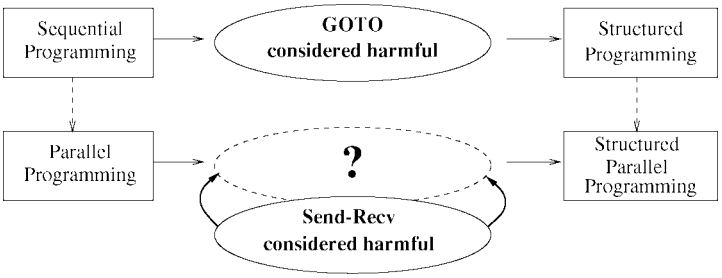

Fig. 1. Just as *goto* complicates sequential programs, *send-receive* causes difficulties in parallel programming.

harmful" [Dijkstra 1968], in which the finger of blame was pointed at the *goto* statement. By that time, Böhm and Jacopini [1966] 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* [Dahl et al. 1975] became almost synonymous with "goto elimination."

If we wish to learn from sequential structured programming, we should answer the question: Which concept or construct plays a similarly negative role to that of *goto* in the parallel setting? As implied in Figure 1 and demonstrated from Section 3 onward, we propose that *send-receive* statements be "considered harmful" and be avoided as far as possible in parallel MPI programs.

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

2. COLLECTIVE OPERATIONS: AN ALTERNATIVE TO SEND-RECEIVE

What would be the proper substitute for *send-receive*? In our view, it does not even need to be invented: we propose using *collective operations*, which are already an established part of MPI.

For the sake of completeness, we show in Figure 2 the main collective operations of MPI for a group of four processes, P0 to P3. The two upper rows of Figure 2 contain collective operations that specify pure communication (*broadcast*, *gather*, etc.); the operations at the bottom of the figure, *reduce* and *scan*, perform both communication and computation. The binary operator specifying computations (+ in Figure 2) is a parameter of the collective operation: it may be either predefined, like addition, or user-defined. If this operator is associative, the corresponding collective operation can be efficiently implemented in parallel.

We address the following challenges to prove the benefits of collective operations as an alternative to send-receive:

—*Simplicity*: Are "collective" programs simpler and more comprehensible?
—*Programmability*: Is a systematic process of program design facilitated?
—*Expressiveness*: Can important classes of applications be conveniently expressed?

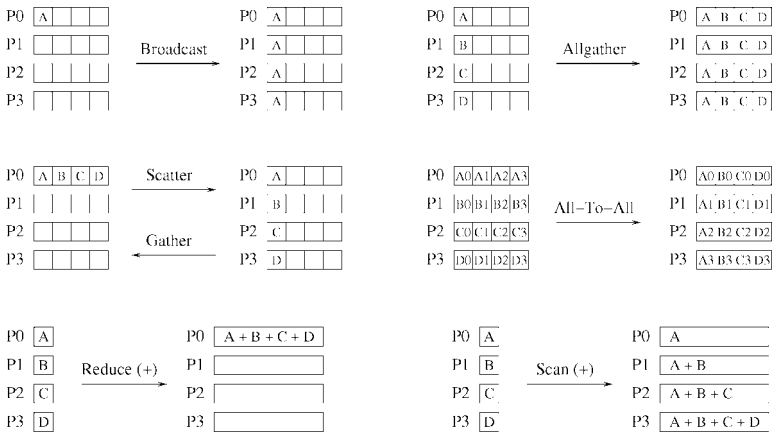

Fig. 2. Collective operations on four processes. Each row of boxes represents data of one process.

—*Performance*: Is performance competitive with that using *send-receive*?

—*Predictability*: Are program behaviour and performance more predictable?

Each of the next sections, one per challenge, opens by stating a commonly held, pro-*send-receive* opinion, which we somewhat polemically call a "myth." We proceed by refuting the myth and conclude with a "reality" statement based on the presented facts and arguments. This "myths-and-realities" structure of the paper enables us to draw a clear conclusion about the suitability of collective operations as an alternative to *send-receive*.

3. 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 need 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, or not. Collective operations 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. Of course, this makes message-passing programming very flexible, but even less comprehensible!

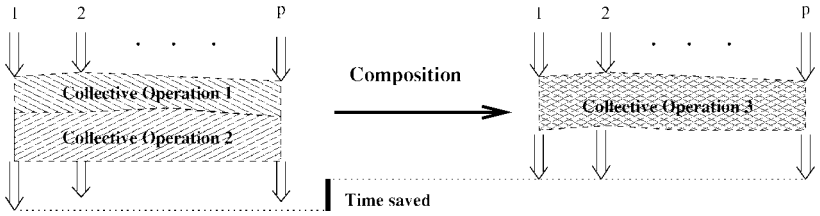

Fig. 3. The idea of fusing two collective operations into one by a transformation like (1).

—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 ([Gorlatch 2001; Pacheco 1997]).

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.

4. 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 collective programs as a sequence of stages facilitates high-level program transformations. One possible kind of transformations fuses two consecutive collective operations into one. This is illustrated in Figure 3 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 Section 7.

One fusion rule 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:

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

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 Gorlatch [2000].

Rule (1) and other rules from Gorlatch [2000] have the advantage that they are (a) proved formally as theorems, (b) parameterized by the occurring operators, for example, `op1` and `op2`, and therefore customizable for a particular application, (c) valid for all possible implementations of collective operations, (d) applicable independently of the parallel target architecture, and (e) suitable for automation.

Table I. 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`

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 Schneider [1997] 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 Gorlatch [2000]) by applying semantically sound transformations like (1). In Section 7, 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.

5. THE CHALLENGE OF EXPRESSIVENESS

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

To refute this quite widely held opinion, we present in Table I 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*.

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 [van de Geijn 1997], which has been implemented entirely without individual communication primitives, as well as by one of the best textbooks on parallel algorithms [Kumar et al. 1994], where the whole methodology centers on implementing and then composing collective operations.

In Fischer and Gorlatch [2002], 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 result of Böhm and Jacopini [1966] for parallel programming.

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

6. THE CHALLENGE OF PERFORMANCE

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

The usual performance argument in favor 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 [Park et al. 1996] 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 [Kielmann et al. 1999]. In Vadhiyar et al. [2000], 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 reimplemented 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: for example, 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 [Bernashi et al. 1999]. 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 nonblocking versions of *send-receive*, `MPI_Isend` and `MPI_Irecv`, are invariably fast owing to the overlap of communication with computation. As demonstrated by Bernashi et al. [1999], in practice 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 smarter and faster algorithms than the collective paradigm. Research has shown that many

Table II. Impact of Transformations on Performance

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

designs using *send-receive* eventually lead to the same high-level algorithms as obtained by the "batch" approach [Goudreau and Rao 1999]. In fact, batch versions often run faster [Hwang and Xu 1998].

—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 in Valiant [1990], 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.

7. THE CHALLENGE OF PREDICTABILITY

Myth: Reliable performance information for parallel programs can only be obtained *a posteriori*, that is, 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 Section 4; we can also estimate the impact of every single transformation on the program's performance. Table II contains a list of transformations from Gorlatch et al. [1999], together with the conditions under which these transformations improve performance.

In Table II, 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-hand column of the table. The estimates were validated in experiments on a Cray T3E and a Parsytec GCel 64 (see Gorlatch [2000] 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 Figure 4.

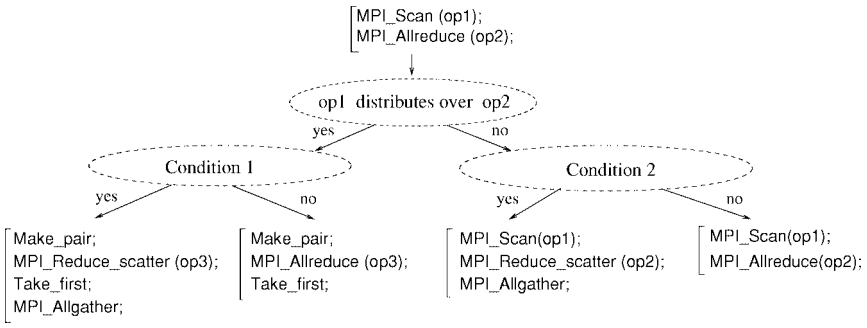

Fig. 4. The tree of design alternatives with decisions made in the nodes.

The best design decision is obtained by checking the design conditions, which depend either on the problem properties, for example, 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 Figure 4. If the block size in an application is small, `Condition 1` (see Gorlatch [2000]) 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 [Kielmann et al. 2000]. However, these models are overly detailed and difficult for an application programmer to use, as demonstrated by a comparison with batch-oriented models [Bilardi et al. 1996; Goudreau et al. 1996].

Reality: Collective operations contribute to the challenging goal of predicting program characteristics during the design process, that is, 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 modeling task at the application level, as compared with models like LogP.

8. 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 have briefly demonstrated 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 have presented hard evidence that many widely held opinions about *send-receive* versus 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 nondeterminism and low-level communication are useful, for example, 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 [1968] famous letter, 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 collective operations have good potential for overcoming this problem and enabling the design of well-structured, efficient, parallel programs based on message passing.

ACKNOWLEDGMENTS

The three anonymous referees did a great job of improving the presentation. 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.

REFERENCES

BERNASHI, M., IANNELLO, G., AND LAURIA, M. 1999. Experimental results about MPI collective communication operations. In *High-Performance Computing and Networking*. Lecture Notes in Computer Science, vol. 1593. Springer-Verlag, Berlin, Germany, 775–783.

BILARDI, G., HERLEY, K., PIETRACAPRINA, A., PUCCI, G., AND SPIRAKIS, P. 1996. BSP vs. LogP. In *Eighth ACM Symposium on Parallel Algorithms and Architectures*. ACM Press, New York, NY, 25–32.

BÖHM, C. AND JACOPINI, G. 1966. Flow diagrams, turing machines and languages with only two formation rules. *Comm. ACM 9*, 366–371.

DAHL, O.-J., DIJKSTRA, E. W., AND A.R.HOARE, C. 1975. *Structured Programming*. Academic Press, New York, NY.

DIJKSTRA, E. W. 1968. Go To statement considered harmful. *Comm. ACM 11*, 3, 147–148.

FISCHER, J. AND GORLATCH, S. 2002. Turing universality of morphisms for parallel programming. *Parallel Process. Lett. 12*, 2.

GORLATCH, S. 2000. Towards formally-based design of message passing programs. *IEEE Trans. Softw. Eng. 26*, 3 (March), 276–288.

GORLATCH, S. 2001. Send-Recv considered harmful? Myths and truths about parallel programming. In *Parallel Computing Technologies (PaCT 2001)*. Lecture Notes in Computer Science, vol. 2127. Springer-Verlag, Berlin, Germany, 243–258.

GORLATCH, S., WEDLER, C., AND LENGAUER, C. 1999. Optimization rules for programming with collective operations. In *Proceedings of IPPS/SPDP'99*, M. Atallah, Ed. IEEE Computer Society Press, Los Alamitos, CA, 492–499.

GOUDREAU, M. ET AL. 1996. Towards efficiency and portablility: programming with the BSP model. In *Proceedings of the Eighth ACM Symposium on Parallel Algorithms and Architectures*. ACM Press, New York, NY, 1–12.

GOUDREAU, M. AND RAO, S. 1999. Single-message vs. batch communication. In *Algorithms for Parallel Processing*, M. Heath, A. Ranade, and R. Schreiber, Eds. Springer-Verlag, Berlin, Germany, 61–74.

GROPP, W., LUSK, E., AND SKJELLUM, A. 1994. *Using MPI: Portable Parallel Programming with the Message Passing*. MIT Press, Cambridge, MA.

HWANG, K. AND XU, Z. 1998. *Scalable Parallel Computing*. McGraw Hill, New York, NY.

KIELMANN, T., BAL, H. E., AND GORLATCH, S. 2000. Bandwidth-efficient collective communication for clustered wide area systems. In *Proceedings of the International Distributed Processing Symposium* (IPDPS 2000). 492–499.

KIELMANN, T. ET AL. 1999. MagPIe: MPI's collective communication operations for clustered wide area systems. In *Proceeding of ACM SIGPLAN PPoPP'99*. ACM Press, New York, NY, 131–140.

KUMAR, V. ET AL. 1994. *Introduction to Parallel Computing*. Benjamin/Cummings, San Fransisco, CA.

PACHECO, P. 1997. *Parallel Programming with MPI*. Morgan Kaufmann, San Fransisco, CA.

PARK, J.-Y. L. ET AL. 1996. Construction of optimal multicast trees based on the parameterized communication model. In *Proceedings of the International Conference on Parallel Processing (ICPP)*. Vol. I. 180–187.

SCHNEIDER, F. B. 1997. *On Concurrent Programming*. Springer-Verlag, Berlin, Germany.

VADHIYAR, S. S., FAGG, G. E., AND DONGARRA, J. 2000. Automatically tuned collective communications. In *Proceedings of Supercomputing 2000* (Dallas, TX).

VALIANT, L. 1990. General purpose parallel architectures. In *Handbook of Theoretical Computer Science*. Vol. A, Chap. 18. MIT Press, Cambridge, MA, 943–971.

VAN DE GEIJN, R. 1997. *Using PLAPACK: Parallel Linear Algebra Package*. MIT Press, Cambridge, MA.

Received November 2001; revised May 2002; accepted February 2003