Optimization Rules for Programming with Collective Operations

Sergei Gorlatch, Christoph Wedler and Christian Lengauer

Fakultät für Mathematik und Informatik, Universität Passau, D-94030 Passau, Germany

Abstract

We study how several *collective operations* like broadcast, reduction, scan, etc. can be composed efficiently in complex parallel programs. Our specific contributions are: (1) a formal framework for reasoning about collective operations; (2) a set of optimization rules which save communications by fusing several collective operations into one; (3) performance estimates, which guide the application of optimization rules depending on the machine characteristics; (4) a simple case study with the first results of machine experiments.

Keywords: collective operations; MPI; communication optimization; performance evaluation; broadcast; reduction; scan.

1 Introduction

Collective operations – broadcast, reduction, scan (parallel prefix), scatter, etc. – describe common patterns of communication and computation in parallel computing. They have become an essential part of parallel languages based on various paradigms and targeting various architectures: conventional SPMD programming with MPI [9] and PVM, bulk-synchronous BSP libraries [11], environments for clusters of SMPs like SIMPLE [3], array languages like HPF [14] and ZPL [16], functional languages like NESL [4], high-level programming environments like P3L [1], intermediate representations in loop parallelization [13], and others.

Collective operations offer several advantages over lower-level constructs like explicit send-receive statements. First, the use of collective operations simplifies the program structure greatly, which makes programming less error-prone. Second, the vendors' activity can be focused on an efficient implementation of a restricted set of collective patterns, either in software or in dedicated hardware, like, e.g., fast reduction on the majority of modern parallel machines. Last but not least, relying on a standard set of collective operations makes the traditionally hard porting and performance prediction of parallel programs easier.

So far, research on collective operations has had two foci. One focus has been on *algorithms*, with the objective of expressing particular classes of applications in terms of collective operations. Recent advances include the PLAPACK library of parallel linear algebra routines [18] and a set of efficient algorithms for computational geometry [5], which are both based on exclusively collective operations, without messing around with individual send-receive statements. The other focus has been on *implementations*, with the objective of realizing particular collective operations efficiently on various parallel architectures. Solutions based on spanning trees, hypercubes, pipelines, as well as hybrid schemes have been reported and benchmarked on available parallel machines [17].

In this paper, we build on the results on algorithms and implementations, and develop a new approach of *performance-directed programming with collective operations*. Our motivation is that a big parallel application may exploit many collective operations, which, consequently, should be used and optimized in combination rather than in isolation.

Our specific objective is to develop a set of optimization rules which, with the premise of some condition, transform a sequence of two or more collective operations into one collective operation which may offer a substantial performance improvement:

$$\texttt{collective op\_1; collective op\_2} \ \stackrel{condition}{\longrightarrow}\ \texttt{collective op\_3}$$

We aim at rules which are *semantic equalities*, i.e., provable in a suitable formalism, and applicable independently of the particular syntax or implementation of collective operations. For illustration purposes, we use an MPI-like syntax. The choice of combinations of collective operations and their optimization in the design process depends usually on how they perform on a given architecture. To support the design process, we augment the set of rules for manipulating (combinations of) collective operations with a mechanism for reliable performance estimates.

The contributions and structure of this paper are as follows. We present a formal framework for reasoning about programs with collective operations (Section 2). We develop a set of optimization rules for compositions of the most popular collective operations: broadcast, reduction and scan (Section 3). We estimate their impact on the target performance (Section 4). And we present a case study with experimental results (Section 5).

2 Programming Model and Formal Framework

In this section, we describe the class of programs under consideration and present a framework, which allows formal reasoning about collective operations.

2.1 Example Program and Sources of Optimization

We consider programs which are composed of two kinds of parallel operations: (1) *local* operations which are computations performed by each processor independently of other processors, and (2) *collective* operations which express interprocessor communication. A collective operation may be either a pure communication as, e.g., broadcast and scatter, or a communication interleaved with computations like in reduction or scan.

As an example of a practically used syntax for collective operations, consider the following program, expressed in a slightly simplified MPI notation:

```
Program Example (x: input, v:output);
  y = f ( x );
  MPI_Scan (y, z, count1, type, op1, comm);
  MPI_Reduce (z, u, count2, type, op2, root, comm);
  v = g ( u );
  MPI_Bcast (v, count3, type, root, comm);
```

Let us look at the structure of program `Example`, leaving a detailed consideration of its individual statements to the next subsection. Program `Example` consists of two local stages, expressed as function calls to `f` and `g`, and three collective stages: scan and reduction (with base operators `op1` and `op2`, correspondingly), and broadcast, all expressed as corresponding MPI calls.

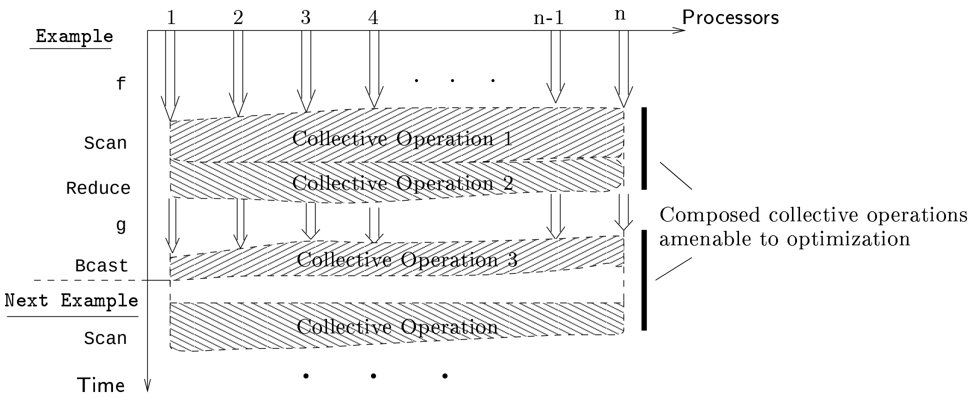

Figure 1: A run time behavior of program `Example` followed by another program, `Next_Example`. Compositions of collective operations are potential optimization points.

Figure 1 demonstrates a possible run time behavior of our example program. As usual in the SPMD style, program `Example` runs on all processors of some parallel machine. In the local stages, each processor follows its own control flow, depicted in the figure by a downward double arrow, independently of the other processors. During collective operations, the control flows of different processors get intertwined in a "common area". Note that local and collective operations may take different amounts of time in the processors, and that we do not require an obligatory synchronization between two subsequent collective operations.

As depicted in Figure 1, compositions of collective operations, which are our main interest, arise in the following cases:

- As part of some individual program, e.g., in `Example`, we have the composition `MPI_Scan (... ); MPI_Reduce (...)`.
- As a consequence of the composition of two programs, e.g., by composing `Example` with another program, `Next_Example`, in one application. If `Next_Example` starts with collective operation `MPI_Scan (... )`, we get the following composition: `MPI_Bcast (...); MPI_Scan (...)`.

Compositions of collective operations can also arise as a result of program transformations if, e.g., some local and collective stages are interchanged, exploiting their data independence.

2.2 Functional Framework

In order to reason about collective operations independently of particular programming models, target architectures and implementations, we develop a common formal framework. Let us use our example for illustration.

Program `Example` starts with input `x` and produces output `v` by computing the following values on the way: $\mathtt{x} \to \mathtt{y} \to \mathtt{z} \to \mathtt{u} \to \mathtt{v}$. We model programs as functions, e.g., function *example* computed by program `Example` transforms input `x` into output `v`:

$$\mathtt{v} \;=\; \mathit{example}\ \mathtt{x} \tag{1}$$

This function is expressed in our formalism by a functional program which is a composition of functions corresponding to the individual statements of `Example`:

$$\mathit{example} \;=\; \mathit{map}\, f \;\;;\;\; \mathit{scan}\,(\otimes) \;\;;\;\; \mathit{reduce}\,(\oplus) \;\;;\;\; \mathit{map}\, g \;\;;\;\; \mathit{bcast} \tag{2}$$

To keep the notation closer to the imperative style of most parallel languages, including MPI, functional composition in program (2) is denoted by a semicolon and is forward, in contrast to the backward composition, traditionally used in the functional world [2, 15]:

$$(f \;\;;\;\; g)\; x \;\;\stackrel{\text{def}}{=}\;\; g\;(f\;x) \tag{3}$$

Function application is denoted by juxtaposition, i.e., $f\,x \;=\; f\,(x)$; it has the highest binding power and associates to the left. The functions representing program stages work on lists: element x_i of list $x \;=\; [x_1, x_2, \ldots, x_n]$ denotes the block of data residing in processor i.

In the computation stage of `Example`, denoted by function call `f(...)`, every processor computes function `f` on its local data, independently of other processors. We model this in program (2) by the higher-order function *map*, which applies unary function f to all elements of a list:

$$map\ f\ [x_1, x_2, \ldots, x_n] \;=\; [\,f\,x_1, f\,x_2,\ \ldots\ , f\,x_n\,] \tag{4}$$

Note that function f in definition (4) may be arbitrarily complex, and that elements $x_1, x_2, \ldots, x_n$ may be lists again.

The collective operations of `Example`, expressed there as MPI functions, are expressed in functional program (2) by functions which we adorn, for simplicity, with only one parameter: the base operator. The input and output parameters, i.e., `x, y,` etc. are omitted in (2) by assuming that the output of each stage is the input of the next stage. Also, we omit the size and the type of the data being communicated, `count` and `type`, which are irrelevant for our program transformations; the size is used in Section 4 for the cost estimates. We assume that all collective operations in a program take place on the same group of processors, so that we can omit the name of the MPI communicator, `comm`. The root processor, `root`, is assumed to be the first processor in the group.

Let us now consider the meaning of collective operations in the example program. Reduction combines data residing on all processors using an associative base operator, $\oplus$, which may be either predefined (addition, multiplication, etc.) or defined by the programmer. We use two versions of reduction, depending on whether the result is assigned to the root or to all processors (in MPI, they are denoted by `MPI_Reduce` and `MPI_Allreduce`, respectively):

$$reduce\,(\oplus)\,[x_1, x_2,\ \ldots\ , x_n] \;=\; [y, x_2, \ldots, x_n] \tag{5}$$

$$allreduce\,(\oplus)\,[x_1, x_2,\ \ldots\ , x_n] \;=\; [y, y,\ \ldots\ , y] \tag{6}$$

where $y \;=\; x_1 \oplus x_2 \oplus \ldots \oplus x_n$.

Another collective operation in `Example` is `MPI_Scan`: on each processor, the result is an accumulation of the data from the processors with smaller rank, using associative operator `op1`. This is expressed by function *scan* with parameter operator $\otimes$:

$$scan\,(\otimes)\,[x_1, x_2, \ldots, x_n] \;=\; [x_1,\ x_1 \otimes x_2,\ \ldots\ ,\ x_1 \otimes x_2 \otimes\ \ldots \otimes x_n] \tag{7}$$

Broadcast `MPI_Bcast` sends a datum or a block residing on the first processor to all other processors. The data of the other processors are not relevant, so we denote them by the underscore in the definition of function *bcast*:

$$bcast\,[x_1, \_,\ \ldots\ , \_] \;=\; [x_1,\ x_1,\ \ldots\ ,\ x_1] \tag{8}$$

The format of program (2) covers also other styles of programming with collective communications, different from MPI. For example, multithreaded computations in the symmetric multiprocessor nodes of clusters of SMPs [3] can be expressed by introducing one more level of parallelism: $map\,(map\,f)$ instead of $map\,f$.

2.3 Auxiliary Variables

Our program transformations are based on introducing auxiliary variables. This technique is ubiquitous in parallel programming practice for removing data dependences by means of duplicating some data and introducing additional computations.

We will use the following functions for data duplication:

$$pair\ a \stackrel{\text{def}}{=} (a, a) \tag{9}$$

$$triple\ a \stackrel{\text{def}}{=} (a, a, a) \tag{10}$$

$$quadruple\ a \stackrel{\text{def}}{=} (a, a, a, a) \tag{11}$$

The next function, π_1, extracts the first element of an arbitrary tuple:

$$\pi_1\,(a, b) \;=\; \pi_1\,(a, b, c) \;=\; \pi_1\,(a, b, c, d) \;=\; a \tag{12}$$

To illustrate the technique of auxiliary variables, consider a simple program term, $P1$, consisting of just one global reduction which sums the elements of a list:

$$P1 \;=\; allreduce\,(+)$$

and another program term, $P2$, which consists of three stages; it creates a list of pairs, computes a reduction on it and then extracts the first element of each pair:

$$P2 \;=\; map\,pair\ ;\ allreduce\,(op\_new)\ ;\ map\,\pi_1$$

where the base operator of reduction, op_new, is defined on pairs, in prefix notation:

$$op\_new\,((a_1, b_1), (a_2, b_2)) \stackrel{\text{def}}{=} (a_1 + a_2, b_1 * b_2)$$

The reduction in $P2$ computes not only the sum of the list elements but also their product. However, $map\,\pi_1$, executed after the reduction, delivers only the sum. Therefore, both $P1$ and $P2$ yield the same output, i.e., they are semantically equivalent.

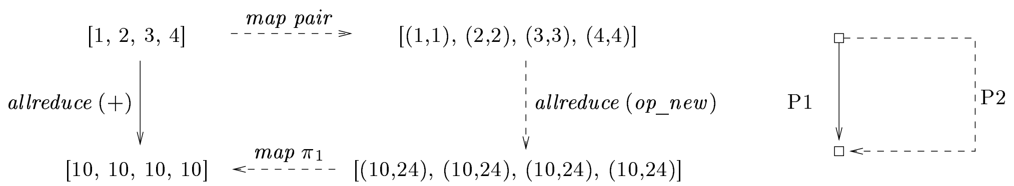

Figure 2: Equivalence of program terms $P1$ and $P2$ on a particular input list.

Semantic equality $P1 = P2$ is illustrated by the diagram in Figure 2 with input list $[1, 2, 3, 4]$ as an example. Due to their equivalence, terms $P1$ and $P2$ can be substituted for each other in an arbitrary parallel program. The cost of program term $P2$ is obviously higher than the cost of $P1$, due to the extra computation and communication in the reduction stage. In the next section we will demonstrate that introducing auxiliary variables can also save costs.

3 Optimization Rules

In this section, we present optimization rules for compositions of several collective operations. We show that, under certain conditions, particular sequences of collective operations can be replaced by a semantically equivalent single collective operation, or even by a local computation on one processor, usually at the price of more complex computations. In other words, we trade expensive communications for comparatively cheap computations. Technically, the extra computations are expressed by the auxiliary functions introduced in Subsection 2.3; our notion of equivalence is illustrated in Figure 2. Formal proofs of most rules can be found in our previous work [7, 19].

3.1 Format of Optimization Rules

We present our optimization rules in the following format:

Rule Name
original program term $\downarrow$ { conditions which must be met to use the rule}   ***optimized program term***
locally defined functions

We distinguish four classes of rules, named after the kind of operation which is the result of the optimization: Reduction, Scan, Comcast (explained further on) and Local. The names of the rules are constructed from the initials of the collective operations in the program term being transformed and the name of the class. If the composed collective operations have different base operations, a 2 appears in the name of the rule. E.g., if a rule transforms a scan (initial "S") with base operation $\otimes$ and a subsequent reduction (initial "R") with a different base operation, $\oplus$, into a reduction, then the rule is called SR2-Reduction.

3.2 Transformations into Reduction

Class Reduction contains transformations which fuse a scan and a subsequent reduction into a single reduction, with some pre- and post-adjustment using auxiliary variables.

The first two rules refer to the case that the base operations in the scan, $\otimes$, and in the reduction, $\oplus$, are different, and $\otimes$ distributes over $\oplus$, i.e., $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$. In the rules, $\oplus$ and $\otimes$ stand for generic binary associative operators.

The corresponding fusion rules for both versions of reduction, *reduce* and *allreduce*, look similar:

SR2-Reduction

$$\begin{array}{l} \boldsymbol{scan}\,(\otimes)\ ;\ [\boldsymbol{all}]\boldsymbol{reduce}\,(\oplus) \\ \quad \big\downarrow \{\otimes \text{ distributes over } \oplus\} \\ \boldsymbol{map}\,\boldsymbol{pair}\ ;\ [\boldsymbol{all}]\boldsymbol{reduce}\,(\boldsymbol{op\_sr2})\ ;\ \boldsymbol{map}\,\boldsymbol{\pi_1} \end{array}$$

$$op\_sr2\ ((s_1, r_1), (s_2, r_2)) \ = \ (s_1 \oplus (r_1 \otimes s_2)\,, r_1 \otimes r_2)$$

A possible impact of the SR2-reduction rule is illustrated in Figure 3: it transforms a scan followed by a reduction, into one reduction, with a possible time saving. The computations expressed by functions *pair* and π_1 (which are explained in Subsection 2.3) is fused in the figure with the corresponding local stages. The performance estimates for this and other rules are given in Section 4.

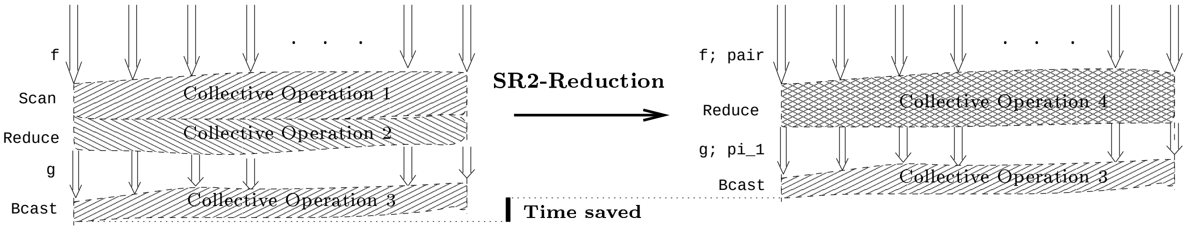

Figure 3: Impact of rule SR2-Reduction on program `Example`.

If the distributivity condition in the SR2-Reduction rule is not met, the fusion of a scan with a reduction is possible only if both have the same base operator which is commutative:

SR-Reduction

$$\begin{array}{l} \boldsymbol{scan}\,(\oplus)\ ;\ [\boldsymbol{all}]\boldsymbol{reduce}\,(\oplus) \\ \quad \big\downarrow \{\oplus \text{ is commutative}\} \\ \boldsymbol{map}\,\boldsymbol{pair}\ ;\ [\boldsymbol{all}]\boldsymbol{reduce\_balanced}\,(\boldsymbol{op\_sr})\ ;\ \boldsymbol{map}\,\boldsymbol{\pi_1} \end{array}$$

$$op\_sr\ ((t_1, u_1), (t_2, u_2)) = (t_1 \oplus t_2 \oplus u_1, uu \oplus uu) \qquad uu = u_1 \oplus u_2$$
$$op\_sr\ ((), (t_2, u_2)) = (t_2, u_2 \oplus u_2)$$

Variable uu is introduced in order to save computations: we need to perform $\oplus$ only four times, rather than five times, during the computation of op_sr.

Unlike the SR2-rule, whose result contains the usual reduction with an associative base operator, the SR-rule requires a particular reduction-like function, called *reduce_balanced*. The reason is that the operator op_sr is, in general, not associative.

We define *reduce_ balanced* using a virtual binary tree with the following conditions:

- all leaves in the tree must have the same depth, i.e., the same path length to the root
- the right subtree of a node must be complete if the node has a non-empty left subtree.

These conditions do not restrict the tree to be complete: there is a (unique) tree for any number of leaves. To illustrate this, our examples here and in the sequel use six processors rather than a power of 2. Horizontal, undirected links express transitions between steps in one processor, directed links denote communications, see Figure 4:

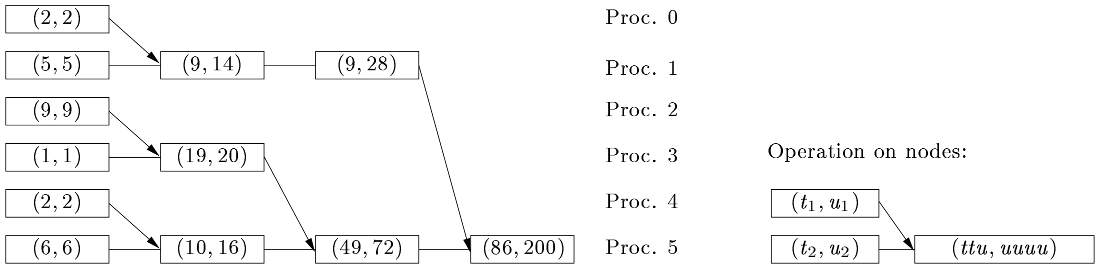

Figure 4: "Balanced reduction" illustrating rule SR-Reduction with $\oplus = +$ (here $ttu = t_1 + t_2$, $uuuu = uu + uu$, $uu = u_1 + u_2$).

The tree can be extended to a butterfly to compute the balanced version of *allreduce*.

3.3 Transformations into Scan

Rules in this class transform a composition of two scans into a single scan. For two scans with different base operators, we require again distributivity:

SS2-Scan

$$\begin{array}{l} \textbf{\textit{scan}}\,(\otimes)\ \ ;\ \ \textbf{\textit{scan}}\,(\oplus) \\ \quad\downarrow \{\ \otimes \text{ distributes over } \oplus\} \\ \textbf{\textit{map pair}}\ \ ;\ \ \textbf{\textit{scan}}\,(\textbf{\textit{op\_sr2}})\ \ ;\ \ \textbf{\textit{map}}\,\boldsymbol{\pi_1} \end{array}$$

$$op\_sr2\ ((s_1, r_1), (s_2, r_2)) \ = \ (s_1 \oplus (r_1 \otimes s_2)\,, r_1 \otimes r_2)$$

For two scans with the same operator, we require commutativity, as well as a specific, balanced scan with a non-associative operator on the right-hand side:

SS-Scan
$\boldsymbol{scan}\,(\oplus)\ ;\ \boldsymbol{scan}\,(\oplus)$ $\quad\downarrow\{\oplus \text{ is commutative}\}$ $\boldsymbol{map\,quadruple}\ ;\ \boldsymbol{scan\_balanced}\,(\boldsymbol{op\_ss})\ ;\ \boldsymbol{map}\,\boldsymbol{\pi_1}$
$op\_ss\,((s_1,t_1,u_1,v_1),(s_2,t_2,u_2,v_2))$ $\quad = ((s_1, ttu, uuuu, vv)\,,\,(s_2 \oplus t_1 \oplus v_1, ttu, uuuu, uu \oplus vv))$ $op\_ss\,((s_1,t_1,u_1,v_1),()) = ((s_1,\_,\_,\_),())$ $ttu = t_1 \oplus t_2 \oplus u_1 \qquad uu = u_1 \oplus u_2 \qquad uuuu = uu \oplus uu \qquad vv = v_1 \oplus v_2$

The additional variables ttu, uu, etc. in the definition of op_ss enable the reuse of computed values, which reduces the computational complexity of the operator from twelve to eight elementary operations, i.e., by one third.

Function *scan_balanced* is implementable using the butterfly as known from the literature [12] and demonstrated by Figure 5.

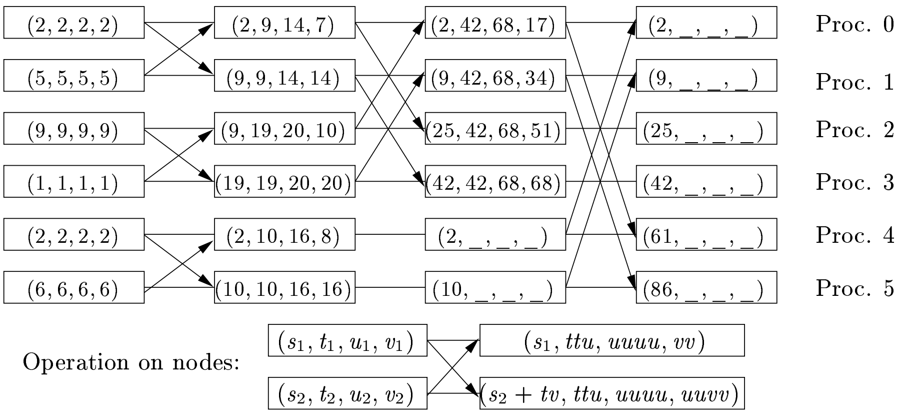

Figure 5: “Balanced scan” illustrating rule SS-Scan with $\oplus = +$ ($tv = t_1 + v_1$, $uuvv = uu + vv$).

If the number of processors is not a power of 2, some processors do not have communication partners. These processors just keep their first value during the corresponding computation step, the other three values are undefined (this is denoted by _ in the rule) and are not used in the subsequent computations.

3.4 Transformations into Comcast

In this section, we consider compositions of a broadcast with one or several scans. Such combinations produce the following target pattern, with function g as parameter. If the root processor holds datum b, then processor i will receive datum $g^i b$, i.e., function g applied i times to b:

$$[b, \_, \ldots, \_] \;\mapsto\; [b, g\,b, \ldots, g^{n-1}\,b]$$

We call this pattern *comcast* (for *com*pute after broad*cast*). Its implementation can broadcast the value b and, subsequently, perform local computations in all processors. A naïve computation in the processors would lead to a linear time complexity:

$$\mathit{bcast}\ ;\ \mathit{map}_{\#}\,(\mathit{times}\ g) \qquad \text{where} \qquad \mathit{times}\ g\ k\ b = g^k$$

Here, function $\mathit{map}_{\#}$ is an extended version of *map* which allows the argument function to have the processor number as an additional parameter:

$$\mathit{map}_{\#} f\,[b_0, b_1, \ldots, b_{n-1}] = [\,f\ 0\ b_0,\ f\ 1\ b_1,\ \ldots,\ f\ (n-1)\ b_{n-1}\,] \tag{13}$$

Let us try to improve the local computation of g^k in processor k to a logarithmic-time algorithm. We use the following schema, with argument functions e and o:

$$\begin{aligned} \mathit{repeat}\,(e, o)\ k\ b \;&=\; \text{if } k = 0 \text{ then } b \text{ else} \\ &\qquad \mathit{repeat}\,(e, o)\,(k \operatorname{div} 2)\,(\text{if } (p \bmod 2 = 0) \text{ then } e\ b \text{ else } o\ b) \end{aligned} \tag{14}$$

Function *repeat* traverses the binary digits of the processor number, k, from the least significant digit to the most significant digit: if the digit is 0, function e is applied, if the digit is 1, function o is applied (see Figure 6 for illustration). Since the number of digits of a number k is $\lceil \log k \rceil$, the time complexity of *repeat* is logarithmic. Note the similarity of this implementation with the method used for the efficient evaluation of powers [10, Sect. 4.6.3].

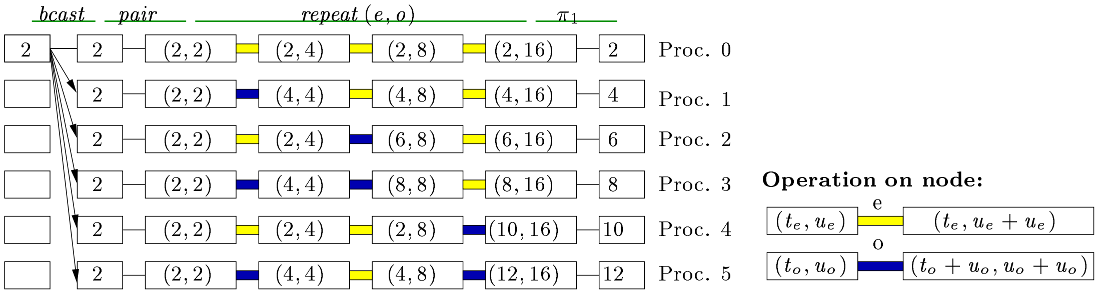

Figure 6: Broadcast and local computations illustrating rule BS-Comcast with $\oplus = +$.

Let us now formulate the rules of group Comcast, for composing broadcast with one or two scans. Pattern *comcast* in these rules, i.e. the computation of g^k, is implemented using a composition of function *repeat* with particular pre- and post-adjustment functions. The adjustment functions and the parameters e and o of pattern *repeat* are determined by the number of scans after the broadcast and by the base operators of the scans.

The first rule fuses a broadcast followed by a scan into a comcast, implemented by a broadcast followed by an adjusted computation using *repeat*, see Figure 6:

BS-Comcast

$$\boldsymbol{bcast} \ ; \ \boldsymbol{scan}\,(\oplus)$$
$$\downarrow\{\ \}$$
$$\boldsymbol{bcast} \ ; \ \boldsymbol{map_{\#}}\ \boldsymbol{op\_comp}$$

$$op\_comp\, k \ = \ pair \ ; \ repeat\,(e,o)\,k \ ; \ \pi_1$$
$$e\,(t,u) = (t, u \oplus u), \qquad o\,(t,u) = (t \oplus u, u \oplus u)$$

The next rule is a corollary of two previous rules, SS2-Scan and BS-Comcast:

BSS2-Comcast

$$\boldsymbol{bcast} \ ; \ \boldsymbol{scan}\,(\otimes) \ ; \ \boldsymbol{scan}\,(\oplus)$$
$$\downarrow\{\ \otimes \text{ distributes over } \oplus\}$$
$$\boldsymbol{bcast} \ ; \ \boldsymbol{map_{\#}}\ \boldsymbol{op\_comp}$$

$$op\_comp\, k \ = \ triple \ ; \ repeat\,(e,o)\,k \ ; \ \pi_1$$
$$e\,(s,t,u) = (s, t \oplus (t \otimes u), u \otimes u) \qquad o\,(s,t,u) = (t \oplus (s \otimes u), t \oplus (t \otimes u), u \otimes u)$$

It would be tempting to obtain also a rule BSS-Comcast as a corollary of SS-Scan and BS-Comcast. Interestingly enough, this does not work: the binary operation used in the SS-Scan is not associative, so that BS-Comcast cannot be applied afterwards. Therefore, rule BSS-Comcast has to be formulated separately:

BSS-Comcast

$$\boldsymbol{bcast} \ ; \ \boldsymbol{scan}\,(\oplus) \ ; \ \boldsymbol{scan}\,(\oplus)$$
$$\downarrow\{\ \oplus \text{ is commutative}\}$$
$$\boldsymbol{bcast} \ ; \ \boldsymbol{map_{\#}}\ \boldsymbol{op\_comp}$$

$$op\_comp\, k \ = \ quadruple \ ; \ repeat\,(e,o)\,k \ ; \ \pi_1$$
$$e\,(s,t,u,v) = (s, t \oplus t \oplus u, uu \oplus uu, v \oplus v) \qquad uu = u \oplus u$$
$$o\,(s,t,u,v) = (s \oplus t \oplus v, t \oplus t \oplus u, uu \oplus uu, uu \oplus v \oplus v)$$

Note that the implementation of *comcast* using *repeat* on all processors is not cost-optimal since, in the first steps, many processors compute the same values. An alternative implementation proceeds as follows: instead of broadcasting the input value b, the first processor computes functions e and o on the input, then it sends the result of o to the second processor; the same is repeated successively with two processors, then four, etc. Interestingly enough, this cost-optimal version yields a worse time complexity than the one based on *repeat*, because of the extra communication overhead for auxiliary variables.

3.5 Transformations into Local

Remember that the local stages of a parallel program are those in which processors work independently of each other, i.e., no collective operation takes place. We demonstrate that some specific compositions of collective operations can actually be replaced by local computations:

BR-Local

$$\begin{array}{l} \boldsymbol{bcast} \ ; \ \boldsymbol{red}\,(\oplus) \\ \quad\downarrow\{\ \} \\ \texttt{iter}\,(\boldsymbol{op\_br}) \end{array}$$

$$op\_br\; s = s \oplus s$$

Here, function iter iterates its argument function $k = \log|xs|$ times on the first element of list xs. The rest is undetermined, while the length of the result is equal to the length of xs:

$$\texttt{iter}\, f\, [x, \_, \ldots, \_] \;=\; [f^{\log|xs|}x, \_, \ldots, \_]$$

Note that the original term broadcasts the first value of the input list to all other processors, whereas the local computation clearly does not do this (as the name "local" suggests). Thus, if the first value is needed in successive computations, either rule BS-Local should not be applied, or the first value should be broadcast additionally (which can still be an optimization). This observation applies also to the other rules of this subsection.

The next rule is derived as a corollary of two previous rules, SR2-Reduction and BR-Local:

BSR2-Local

$$\begin{array}{l} \boldsymbol{bcast} \ ; \ \boldsymbol{scan}\,(\otimes) \ ; \ \boldsymbol{red}\,(\oplus) \\ \quad\downarrow\{\otimes \text{ distributes over } \oplus\} \\ \boldsymbol{map\,pair} \ ; \ \texttt{iter}\,(\boldsymbol{op\_bsr2}) \ ; \ \boldsymbol{map}\,\boldsymbol{\pi_1} \end{array}$$

$$op\_bsr2\,(s,t) = (s \oplus (s \otimes t), t \otimes t)$$

Deriving rule BSR-Local as a corollary of SR-Reduction and BR-Local does not work, because the binary operation used in the result of SR-Reduction is not associative.

BSR-Local

$$\begin{array}{l} \boldsymbol{bcast} \ ; \ \boldsymbol{scan}\,(\oplus) \ ; \ \boldsymbol{red}\,(\oplus) \\ \quad\downarrow\{\oplus \text{ is commutative}\} \\ \boldsymbol{map\,pair} \ ; \ \texttt{iter}\,(\boldsymbol{op\_bsr}) \ ; \ \boldsymbol{map}\,\boldsymbol{\pi_1} \end{array}$$

$$op\_bsr\,(t,u) = (t \oplus t \oplus u, uu \oplus uu) \qquad uu = u \oplus u$$

If the last subject of the composition is *allreduce* instead of *reduce*, all transformation rules into Local can also be used: just broadcast the value of the result. As an example, we change the transformation of BR-Local accordingly:

CR-Alllocal
$\boldsymbol{bcast}\ ;\ \boldsymbol{allred}\,(\oplus)$ $\big\downarrow\{\,\}$ $\texttt{iter}\,(\boldsymbol{op\_br})\ ;\ \boldsymbol{bcast}$
$op\_br\; s = s \oplus s$

4 Performance Estimates

In this section, we estimate the performance impact of the transformation rules presented in the previous section, i.e., we are interested in the conditions under which these rules actually improve the target performance.

4.1 Machine and Implementation Model

In accordance with the trends in modern parallel machines and communication environments, we assume a virtual, fully connected system in which each processor can communicate with any other processor at the same cost. Communication links are bidirectional: two processors can send blocks of size m to each other simultaneously in time

$$T_{send\_recv} \;=\; t_s + m \cdot t_w$$

where t_s is the start-up time and t_w is the per-word transfer time. The time of one computation operation is assumed as unit.

The influence of optimization rules on performance depends also on how the involved collective operations are implemented. We present here estimates for implementations which are mentioned in the literature as most widely used and which are in fact exploited in the version of MPI available to us. This enables a comparison of our estimates with experimental results. If a different software implementation or dedicated hardware is used, the cost estimation must be repeated.

All three collective operations involved in our optimization rules – broadcast, reduction and scan – are implementable using the same, butterfly-like communication pattern [6, 12]. It proceeds in $\log p$ phases: in each phase segments of length m are exchanged pairwise between processors. The only difference is in the computations: there are no computations in broadcast, reduction has one base operation per element and scan has two base operations. Therefore, the time estimates for collective operations with the butterfly implementation are as follows:

$$T_{bcast} = \log p \cdot (t_s + m \cdot t_w) \tag{15}$$

$$T_{reduce} = \log p \cdot (t_s + m \cdot (t_w + 1)) \tag{16}$$

$$T_{scan} = \log p \cdot (t_s + m \cdot (t_w + 2)) \tag{17}$$

4.2 Sample Estimation for SS2-Scan Rule

Let us now illustrate how the impact of an optimization rule on the target performance can be estimated. We consider rule SS2-Scan. Its left-hand side requires the following time implied by equality (17):

$$2 \cdot \log p \cdot (t_s + m \cdot (t_w + 2))$$

The right-hand side performs pairing at the beginning and projection at the end: they contribute just a small additive constant to the whole time, which we ignore. The major part is a scan but, this time, on a list of pairs, with the base operator defined in the rule formulation, which requires time:

$$\log p \cdot (t_s + m \cdot (2 \cdot t_w + 6))$$

Now, it is easy to calculate when the SS2-Scan optimization pays off:

$$t_s + 2 \cdot m \cdot t_w + 6 \cdot m \ < \ 2 \cdot t_s + 2 \cdot m \cdot t_w + 4 \cdot m$$

which simplifies to

$$t_s \ > \ 2 \cdot m$$

Thus, if scan is implemented on a virtual butterfly, then rule SS2-Scan should be applied if the machine has a relatively high start-up cost and/or if the length of the blocks held in the processors is small. This result is in line with intuition: the rule trades synchronization costs, expressed by the start-up, for additional computations which increase with the block size. Note that, for large blocks, rule SS2-Scan may become impractical because of the additional memory consumption.

4.3 Summary of Estimates

We summarize the performance estimates for all rules formulated so far in Table 1. For each rule, we state the time required by its left-hand side, i.e., before the rule application, and the time after applying the rule. A factor of $\log p$ appears in all estimates; we omit it in the table entries and state it in the column headings instead. From the times for the left- and right-hand sides of a rule, we formulate the condition under which the rule improves the target performance. We enter "always" if, for the butterfly implementation, the rule improves the performance independently of the machine parameters.

Rule name	(time before) $\cdot$ $\log p$	(time after) $\cdot$ $\log p$	Improved if
SR2-Reduction	$2t_s + m \cdot (2t_w + 3)$	$t_s + m \cdot (2t_w + 3)$	always
SR-Reduction	$2t_s + m \cdot (2t_w + 3)$	$t_s + m \cdot (2t_w + 4)$	$t_s > m$
SS2-Scan	$2t_s + m \cdot (2t_w + 4))$	$t_s + m \cdot (2t_w + 6)$	$t_s > 2m$
SS-Scan	$2t_s + m \cdot (2t_w + 4)$	$t_s + m \cdot (3t_w + 8)$	$t_s > m(t_w + 4)$
BS-Comcast	$2t_s + m \cdot (2t_w + 2)$	$t_s + m \cdot (t_w + 2)$	always
BSS2-Comcast	$3t_s + m \cdot (3t_w + 4)$	$t_s + m \cdot (t_w + 5)$	$t_w + \frac{1}{m} t_s > \frac{1}{2}$
BSS-Comcast	$3t_s + m \cdot (3t_w + 4)$	$t_s + m \cdot (t_w + 8)$	$t_w + \frac{1}{m} t_s > 2$
BR-Local	$2t_s + m \cdot (2t_w + 1)$	m	always
BSR2-Local	$3t_s + m \cdot (3t_w + 3)$	$3m$	always
BSR-Local	$3t_s + m \cdot (3t_w + 3)$	$4m$	$t_w + \frac{1}{m} \cdot t_s \geq \frac{1}{3}$

Table 1: Performance estimates of optimization rules

5 Case Study: Polynomial Evaluation

In this section, we present a simple case study. Its main objective is to demonstrate the methodical use of the presented optimization rules. We do not claim the optimality of the obtained solutions. We have another example in which rule SR2-Reduction, applied twice, leads to an optimal parallel algorithm, by improving the asymptotic complexity of the initial specification [8].

We consider the problem of evaluating a polynomial,

$$a_1 * x + a_2 * x^2 + \cdots + a_n * x^n$$

on m points: $y_1, \ldots, y_m$. Let us assume, without loss of generality, that the list of coefficients, as, is distributed across the processors such that processor i keeps value a_i, and that the list of values $ys = [y_1, \ldots, y_m]$ is stored in the first processor.

5.1 Program Design

We start with the following obvious parallel program for the polynomial evaluation, which consists of four stages:

$$PolyEval\_1 \ = \ bcast \ ; \ scan\,(*) \ ; \ map2\,(*)\ as \ ; \ reduce\,(+) \tag{18}$$

This program (it could be rather viewed as a specification of polynomial evaluation) takes list ys as input and proceeds as follows:

- $bcast$ broadcasts ys to all other processors;
- $scan\,(*)$ computes $y^i = [y_1^i, \ldots, y_m^i]$ in processor i, $i = 1, \ldots, n$;
- $map2$ is a version of map defined on two lists of equal size: in program (18), stage $map2\,(*)\; as$ means computing in each processor, i, the list:

$$[a_i * y_1^i, \ldots, a_i * y_m^i]$$

- $map2_{\#}$ is a version of $map_{\#}$ defined on two lists of equal size;
- finally, $reduce\,(+)$ sums thus obtained lists in the processors elementwise and puts the result list into the first processor:

$$[\,(\sum_{i=1}^{n} a_i * y_1^i\,)\,, \ldots, \,(\sum_{i=1}^{n} a_i * y_m^i\,)\,]$$

We observe that program (18) exploits three collective operations, the first two of which can be transformed using rule BS-Comcast. This rule applies without requiring any special conditions except associativity of $*$ which can be assumed. Furthermore, the performance estimate in Table 1 guarantees that rule BS-Comcast always provides a performance improvement.

The instantiation of the rule in our case reads as follows:

$$bcast \;;\; scan\,(*) \quad \longrightarrow \quad bcast \;;\; map_{\#}\; op\_poly$$

where operation op_poly on the right-hand side is obtained from generic operator op_comp in the rule by substituting our concrete operator, $*$:

$$op\_poly\; k \;=\; pair \;;\; repeat\,(e, o)\; k \;;\; \pi_1$$
$$e\,(t, u) = (t, u * u), \qquad o\,(t, u) = (t * u, u * u)$$

After applying rule BS-Comcast, the program reads as follows:

$$PolyEval\_2 \;=\; bcast \;;\; map_{\#}\; op\_poly \;;\; map2\,(*)\; as \;;\; reduce\,(+) \tag{19}$$

In (19), two local stages are executed in sequence: $map_{\#}$ and $map2$. By defining a new operation op_new,

$$op\_new\; k\; x\; y = (op\_poly\; k\; x) * y$$

we can fuse them into one local stage in the following target program:

$$PolyEval\_3 \;=\; bcast \;;\; map2_{\#}\,(op\_new\; as) \;;\; reduce\,(+) \tag{20}$$

5.2 First Experimental Results

We conducted first experiments with some of the presented rules on a parallel machine. For rule BS-Comcast, the results on our Parsytec 64-processor network with MPICH 1.0 are presented in the following two plots:

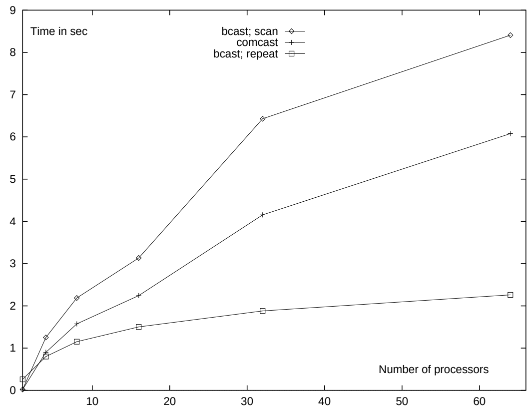

Figure 7: Run time improvement after application of rule BS-Comcast on block size $32 * 10^3$

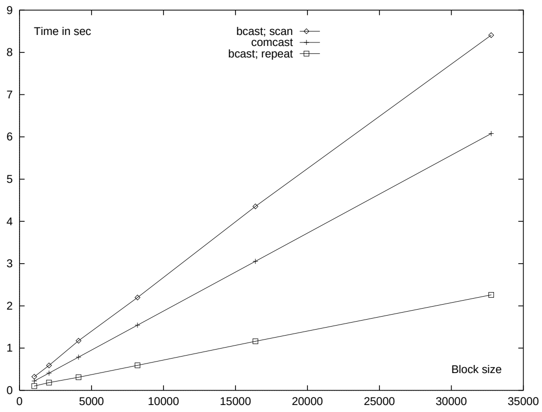

Figure 8: Run time improvement after application of rule BS-Comcast on 64 processors

Here, “bcast; scan” corresponds to the left-hand side of the rule; for the right-hand side, we compare two possibilities discussed in Subsection 3.4: “comcast” corresponds to the cost-optimal version, and “bcast; repeat” corresponds to the more efficient version, demonstrated in Figure 6, which we used in all rules of group Comcast.

Figure 7 shows the dependence of run time on the number of processors, for a fixed block size; Figure 8 shows the dependence on the block size. The experiments confirm that the rule indeed improves the run time as indicated in Table 1. For more experimental results, see [8].

6 Conclusions and Discussion

Collective operations are a convenient means of specifying parallelism. Our theoretical considerations and experimental results demonstrate that it pays to target their combinations for thoughtful and thorough programming. Good optimization here may pay a lot.

Programming with collective operations deserves scrutiny and requires a method. We have sketched such a method, based on a set of transformation rules, augmented with a cost calculus. We have chosen the paradigm of functional programming to prove our rules formally, even though the final implementations will most likely be imperative. Functional framework offers effective support in prototyping and analysis, and does not limit the choice of the final implementation.

We can distinguish between two groups of rules. *General rules* refer to the traditional collective operations known from the literature. This group includes rules SR2-Reduction, SS2-Scan and all Local and Comcast rules. Therefore, these rules can be used directly for optimizing programs that rely on MPI, PVM, SIMPLE and other parallel interfaces. *Special rules* exploit new collective operations like *reduce_balanced*, *scan_balanced*, etc. A rule from this group can be used only if the corresponding collective operation is implemented on a particular machine. The implementation of new operations is a topic of our current research.

One reasonable question to ask is whether the search for combinations of collective operations is open-ended. We have attempted to be exhaustive to a point. In previous work we have identified map, broadcast, reduction and scan, as basic building blocks for linear recursions on lists [20]. When looking at their input/output behavior, which reveals that broadcast is a one-to-all, reduction an all-to-one and scan an all-to-all operation, some combinations can be dismissed as not useful. We have not considered transformation rules for longer than triple combinations. They would require further algebraic properties of the base operator, additionally to associativity and distributivity/commutativity, which makes them less likely to be widely applicable.

The advantage of our formal approach is that the rules can be proved and that they are independent of the way in which the collective operations are implemented on the particular machine. The implementation must be taken into account when estimating the target efficiency.

We have presented a very simple case study and only preliminary experimental results. Another example demonstrates the use of rule SR2-Reduction in the design of an asymptotically optimal algorithm [7]. More studies and experiments with real applications are necessary to understand how far the programming with collective operations can contribute to the success of parallelism.

7 Acknowledgements

We are grateful to Holger Bischof for suggesting us the case study, and for his help in the computer experiments. Thanks are due to Peter Sanders for a fruitful discussion on the implementation of the comcast pattern, and also to Marco Aldinucci, Christoph A. Herrmann and Susanna Pelagatti for helpful comments on the manuscript.

References

[1] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^{3}L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[2] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM*, 21(8):613–642, Aug. 1978.

[3] D. Bader and J. JáJá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). Technical Report CS-TR-3798/UMIACS-TR-97-48, Department of Computer Science, University of Maryland at College Park, May 1997.

[4] G. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, Department of Computer Science, Carnegie-Mellon University, 1992.

[5] X. Deng and N. Gu. Good programming style on multiprocessors. In *Proc. IEEE Symp. on Parallel and Distributed Processing (SPDP'94)*, pages 538–543. IEEE Computer Society Press, 1994.

[6] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Parallel Processing. Euro-Par'96, Vol. II*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[7] S. Gorlatch. Optimizing compositions of scans and reductions in parallel program derivation. Technical Report MIP-9711, Universität Passau, May 1997. Available at http://www.fmi.uni-passau.de/cl/papers/Gor97b.html.

[8] S. Gorlatch and C. Lengauer. Homomorphism (de)compositions for parallel programming. In *Proc. 3rd Int. Working Conf. on Massively Parallel Programming Models (MPPM'97)*, pages 23–32. IEEE Computer Society Press, 1998.

[9] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* Scientific and Engineering Computation Series. MIT Press, 1994.

[10] D. E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms.* Addison-Wesley, 2nd edition, 1980.

[11] W. F. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science 1000, pages 46–61. Springer-Verlag, 1995.

[12] M. J. Quinn. *Parallel Computing.* McGraw-Hill, 1994.

[13] X. Redon and P. Feautrier. Detection of reductions in sequential programs with loops. In A. Bode, M. Reeve, and G. Wolf, editors, *Parallel Architectures and Languages Europe (PARLE'93)*, Lecture Notes in Computer Science 694, pages 132–145. Springer-Verlag, 1993.

[14] R. Schreiber. High Performance Fortran, Version 2. *Parallel Processing Letters*, 7(4):437–449, 1997.

[15] D. B. Skillicorn. *Foundations of Parallel Programming.* Cambridge International Series on Parallel Computation. Cambridge University Press, 1994.

[16] L. Snyder. A ZPL programming guide (Version 6.2). Technical report, Department of Computer Science, University of Washington, Jan. 1998.

[17] R. van de Geijn. On global combine operations. *J. Parallel and Distributed Computing*, 22:324–328, 1994.

[18] R. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package.* Scientific and Engineering Computation Series. MIT Press, 1997.

[19] C. Wedler and C. Lengauer. Parallel implementations of combinations of broadcast, reduction and scan. In G. Agha and S. Russo, editors, *Proc. 2nd Int. Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'97)*, pages 108–119. IEEE Computer Society Press, 1997.

[20] C. Wedler and C. Lengauer. On linear list recursion in parallel. *Acta Informatica*, 35(10):875–909, 1998.