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 machine experiments.*

1. Introduction

Collective operations – broadcast, reduction, scan, 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 run-time libraries like MPI [7] and BSP [8], environments for clusters of SMPs like SIMPLE [2], array languages like HPF [11] and ZPL [12], functional languages like NESL [3], high-level programming environments like P3L [1], intermediate representations in loop parallelization [10], and others.

Collective operations offer several advantages over explicit send-receive statements. First, the use of collective operations simplifies the program structure, 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. Last but not least, the use of collective operations simplifies the traditionally hard porting and performance prediction of parallel programs.

So far, research has concentrated either on expressing *applications* in terms of collective operations (see, e.g., the PLAPACK library for linear algebra [14] or algorithms for computational geometry [4], both based on exclusively collective communications), or on efficient *implementations* of particular collective operations (see, e.g., [13]).

In this paper, we build on the results on applications and implementations, and develop a new approach of *performance-directed programming with collective operations*. Our motivation is that a big application may exploit many collective operations, which 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, can be used to transform a sequence of two or more collective operations into one collective operation, with the possible gain of a substantial performance improvement:

$$\texttt{coll_op_1; coll_op_2} \stackrel{cond}{\longrightarrow} \texttt{coll_op_3}$$

We aim at rules which are *semantic equalities*, i.e., provable in a suitable formalism, and applicable independently of the particular implementation of collective operations. For illustration purposes, we use an MPI-like syntax. 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 Formalism

2.1. Example Program and Optimization Sources

We consider SPMD 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 or 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 SPMD 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);
```

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.

Figure 1 demonstrates how program `Example` runs on some parallel machine. In the local stages, each processor follows its own control flow, depicted by a downward double arrow. During collective operations, the control flows of the 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 a synchronization between collective operations.

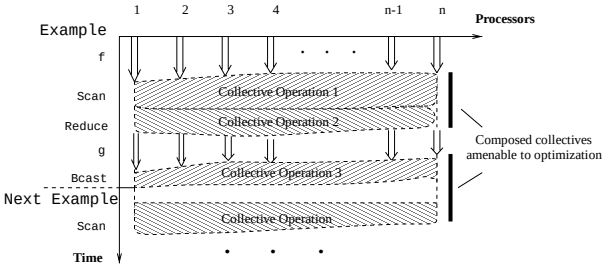

Figure 1. Run time behavior of program `Example`, followed by program `Next_Example`. Compositions of collective operations are potential optimization points.

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

- As part of a program, e.g.,in `Example` the composition `MPI_Scan (...); MPI_Reduce (...)`.
- As a result of composing, e.g., `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 (...)`.

2.2. Functional Framework

To enable formal reasoning about collective operations, we use the functional view of programs. For instance, our program `Example` takes input `x` and produces output `v`, by computing the following values: `x` $\rightarrow$ `y` $\rightarrow$ `z` $\rightarrow$ `u` $\rightarrow$ `v`. In other words, program `Example` computes function ex, which transforms input `x` into output `v`:

$$\texttt{v} \;=\; ex\ \texttt{x} \tag{1}$$

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

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

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:

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

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:

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

Here, function f may be arbitrarily complex and elements $x_1, x_2, \ldots, x_n$ may be lists (blocks) 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. We omit the size and the type of the data, `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):

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

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

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

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 an associative base operator. This is expressed in (2) by function $scan$ with parameter operator $\otimes$:

$$scan\,(\otimes)\,[x_1, x_2, \ldots, x_n] \stackrel{\text{def}}{=} [x_1,\ x_1 \otimes x_2,\ \ldots,\ x_1 \otimes x_2 \otimes \cdots \otimes x_n] \quad (7)$$

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 $bcast$:

$$bcast\,[x_1, _, \ldots, _] \stackrel{\text{def}}{=} [x_1,\ x_1,\ \ldots,\ x_1] \quad (8)$$

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

2.3. The Technique of Optimization

Our program transformations are based on introducing auxiliary variables, which is a ubiquitous technique in parallel programming practice for removing data dependences.

We use the following functions for data duplication:

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

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

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

Function π_1 extracts the first element of an arbitrary tuple:

$$\pi_1\,(a, b) = \pi_1\,(a, b, c) = \pi_1\,(a, b, c, d) \stackrel{\text{def}}{=} a \quad (12)$$

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 in $P2$, 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, $P2$ always yields the same output as $P1$, i.e., they are semantically equivalent.

The semantic equality $P1 = P2$ is illustrated by the diagram in Figure 2, with input list $[1, 2, 3, 4]$. In this example, 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 incur a significant saving of costs.

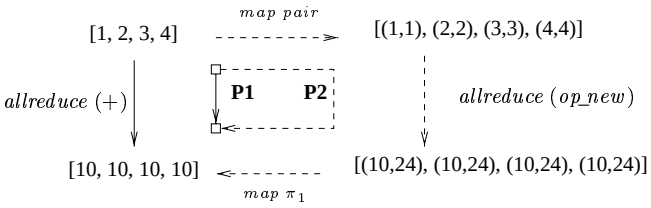

Figure 2. Equivalence of programs $P1$ and $P2$.

3. Optimization Rules

In this section, we show how particular sequences of collective operations can be replaced by a semantically equivalent single collective operation, or even by local computations, 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. Formal proofs of most rules can be found in our previous work [6, 15].

3.1. Format of Optimization Rules

We present optimization rules in the following format:

Rule Name
$original\ program\ term$ ↓ {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, $\oplus$, and the result of the transformation is one reduction, then the rule is called SR2-Reduction.

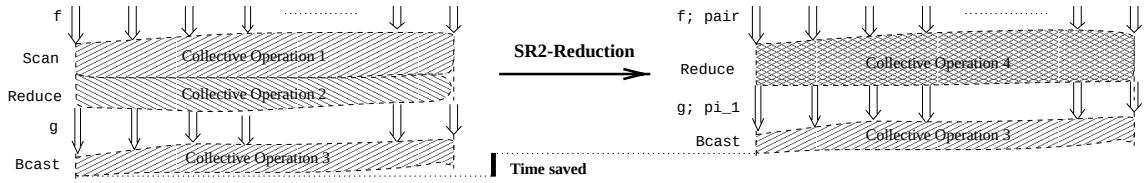

Figure 3. Impact of rule SR2-Reduction on program `Example`.

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.

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 fusion rules for both versions of reduction, $reduce$ and $allreduce$, look similar (see [6] for a proof):

SR2-Reduction

$$\boldsymbol{scan}\ (\otimes)\ ;\ [\boldsymbol{all}]\boldsymbol{reduce}\ (\oplus)$$

$$\downarrow \{\otimes \text{ distributes over } \oplus\}$$

$$\boldsymbol{map\,pair}\ ;\ [\boldsymbol{all}]\boldsymbol{reduce}\ (\boldsymbol{op_sr2})\ ;\ \boldsymbol{map}\ \boldsymbol{\pi_1}$$

$$op_sr2\ ((s_1, r_1), (s_2, r_2)) = (s_1 \oplus (r_1 \otimes s_2), r_1 \otimes r_2)$$

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 (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.

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

SR-Reduction

$$\boldsymbol{scan}\ (\oplus)\ ;\ [\boldsymbol{all}]\boldsymbol{reduce}\ (\oplus)$$

$$\downarrow \{\oplus \text{ is commutative}\}$$

$$\boldsymbol{map\,pair}\ ;\ [\boldsymbol{all}]\,\boldsymbol{reduce_bal}\ (\boldsymbol{op_sr})\ ;\ \boldsymbol{map}\ \boldsymbol{\pi_1}$$

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

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 yields a reduction-like function *reduce_bal*, whose operator, op_sr, may be non-associative. We define *reduce_bal* using a virtual binary tree (see Figure 4), whose leaves all have the same path length to the root, and a right subtree must be complete if its root has a non-empty left subtree. The whole tree does not have to be complete: our example shows six processors, rather than a power of 2:

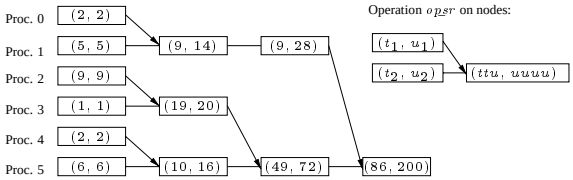

Figure 4. "Balanced reduction" illustrating rule SR-Reduction with $\oplus = +$ (here $ttu = t_1 + t_2 + u_1$, $uuuu = uu + uu$, $uu = u_1 + u_2$).

Horizontal, undirected links express transitions between steps in one processor, directed links denote communications. 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

$$\boldsymbol{scan}\ (\otimes)\ ;\ \boldsymbol{scan}\ (\oplus)$$

$$\downarrow \{\otimes \text{ distributes over } \oplus\}$$

$$\boldsymbol{map\,pair}\ ;\ \boldsymbol{scan}\ (\boldsymbol{op_sr2})\ ;\ \boldsymbol{map}\ \boldsymbol{\pi_1}$$

$$op_sr2\ ((s_1, r_1), (s_2, r_2)) = (s_1 \oplus (r_1 \otimes s_2), r_1 \otimes r_2)$$

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

$$\boldsymbol{scan}\ (\oplus)\ ;\ \boldsymbol{scan}\ (\oplus)$$

$$\downarrow \{\oplus \text{ is commutative}\}$$

$$\boldsymbol{map\,quadruple}\ ;\ \boldsymbol{scan_bal}\ (\boldsymbol{op_ss})\ ;\ \boldsymbol{map}\ \boldsymbol{\pi_1}$$

$$op_ss\ ((s_1, t_1, u_1, v_1), (s_2, t_2, u_2, v_2)) =$$
$$((s_1, ttu, uuuu, vv)\ ,\ (s_2 \oplus t_1 \oplus v_1, ttu, uuuu, uu \oplus vv))$$
$$op_ss\ ((s_1, t_1, u_1, v_1), ()) = ((s_1, _, _, _), ())\ ,\ vv = v_1 \oplus v_2$$
$$ttu = t_1 \oplus t_2 \oplus u_1\ ,\ uu = u_1 \oplus u_2\ ,\ uuuu = uu \oplus uu$$

Function $scan_bal$ is implementable using the butterfly demonstrated by Figure 5. If the number of processors is not a power of 2, some processors do not have communication partners; they keep their first value during the corresponding step, the other three values are undefined (denoted by _) and are not used in the subsequent computations.

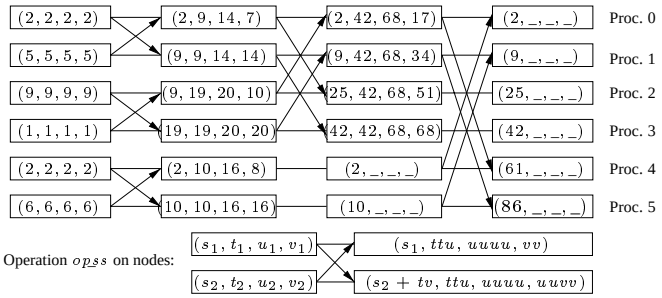

Figure 5. "Balanced scan" illustrating rule SS-Scan with $\oplus = +$, where $tv = t_1 + v_1$, $uuvv = uu + vv$.

3.4. Transformations into Comcast

Compositions of a broadcast with one or several scans 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 element b: $[b, _, \ldots, _] \;\mapsto\; [b, g\ b, \ldots, g^{n-1}\ b]$.

We call this pattern $comcast$ (for compute after broad$cast$). A naïve computation in the processors would lead to a linear time complexity:

$$bcast\ ;\ map_{\#}\ (times\ g), \qquad \text{where} \qquad times\ g\ k\ =\ g^k$$

Here, function $map_{\#}$ is a map which allows the argument function to have the processor number as parameter:

$$map_{\#}\ f\ [b_0, b_1, \ldots, b_{n-1}] \stackrel{\text{def}}{=} [f\ 0\ b_0,\ f\ 1\ b_1,\ \ldots,\ f\ (n-1)\ b_{n-1}] \quad (13)$$

We can improve the computation of g^k in processor k to a logarithmic-time algorithm, using the following schema, with argument functions e and o:

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

This schema is used in our first rule of group Comcast, which fuses a broadcast followed by a scan into a comcast:

BS-Comcast

$$\begin{array}{l} \boldsymbol{bcast}\ ;\ \boldsymbol{scan}\ (\oplus) \\ \quad\downarrow\{\} \\ \boldsymbol{bcast}\ ;\ \boldsymbol{map_{\#}}\ \boldsymbol{op_comp} \end{array}$$

$$\begin{array}{c} op_comp\ k\ =\ pair\ ;\ repeat\,(e, o)\ k\ ;\ \pi_1 \\ e\,(t, u) = (t, u \oplus u), \qquad o\,(t, u) = (t \oplus u, u \oplus u) \end{array}$$

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). Note the similarity of this implementation with the well-known method of the efficient evaluation of powers.

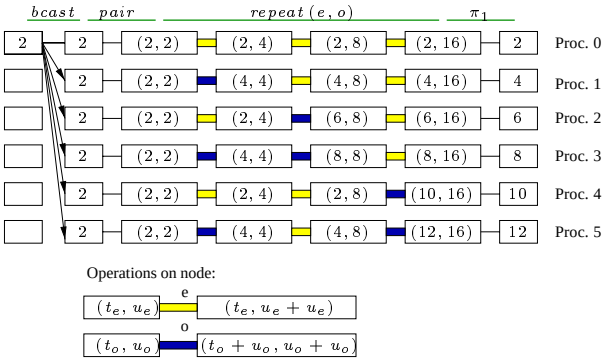

Figure 6. Rule BS-Comcast with $\oplus = +$.

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

BSS2-Comcast

$$\begin{array}{l} \boldsymbol{bcast}\ ;\ \boldsymbol{scan}\ (\otimes)\ ;\ \boldsymbol{scan}\ (\oplus) \\ \quad\downarrow\{\otimes \text{ distributes over } \oplus\} \\ \boldsymbol{bcast}\ ;\ \boldsymbol{map_{\#}}\ \boldsymbol{op_comp} \end{array}$$

$$\begin{array}{c} op_comp\ k\ =\ triple\ ;\ repeat\,(e, o)\ k\ ;\ \pi_1 \\ e\,(s, t, u) = (s, t \oplus (t \otimes u), u \otimes u) \\ o\,(s, t, u) = (t \oplus (s \otimes u), t \oplus (t \otimes u), u \otimes u) \end{array}$$

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

BSS-Comcast

$$\begin{array}{l} \boldsymbol{bcast}\ ;\ \boldsymbol{scan}\ (\oplus)\ ;\ \boldsymbol{scan}\ (\oplus) \\ \quad\downarrow\{\oplus \text{ is commutative}\} \\ \boldsymbol{bcast}\ ;\ \boldsymbol{map_{\#}}\ \boldsymbol{op_comp} \end{array}$$

$$\begin{array}{c} op_comp\ k\ =\ quadruple\ ;\ repeat\,(e, o)\ k\ ;\ \pi_1 \\ e\,(s, t, u, v) = (s, t \oplus t \oplus u, uu \oplus uu, v \oplus v)\ ,\ uu = u \oplus u \\ o\,(s, t, u, v) = (s \oplus t \oplus v, t \oplus t \oplus u, uu \oplus uu, uu \oplus v \oplus v) \end{array}$$

Note that the implementation of $comcast$ using $repeat$ is not cost-optimal: in the first steps, many processors compute the same values. An alternative implementation is 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. However, 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

Transformations of this class replace some specific compositions of collective operations by local computations:

BR-Local

$$bcast\ ;\ red\ (\oplus)$$
$$\downarrow\{\}$$
$$iter\ (op_br)$$

$op_br\ s = s \oplus s$

BR-Alllocal

$$bcast\ ;\ allred\ (\oplus)$$
$$\downarrow\{\}$$
$$iter\ (op_br)\ ;\ bcast$$

$op_br\ s = s \oplus s$

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:

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

If the last subject of the composition is $allreduce$ instead of $reduce$, this and the subsequent transformation rules into Local can also be used: just broadcast the value of the result. 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

$$bcast\ ;\ scan\ (\otimes)\ ;\ red\ (\oplus)$$
$$\downarrow\{\otimes \text{ distributes over } \oplus\}$$
$$map\ pair\ ;\ iter\ (op_bsr2)\ ;\ map\ \pi_1$$

$op_bsr2\ (s,t) = (s \oplus (s \otimes t), t \otimes t)$

Finally, we formulate the rule BSR-Local:

BSR-Local

$$bcast\ ;\ scan\ (\oplus)\ ;\ red\ (\oplus)$$
$$\downarrow\{\oplus \text{ is commutative}\}$$
$$map\ pair\ ;\ iter\ (op_bsr)\ ;\ map\ \pi_1$$

$op_bsr\ (t,u) = (t \oplus t \oplus u, uu \oplus uu) \qquad uu = u \oplus u$

4. Performance Estimates

In this section, we are interested in the conditions under which the rules presented in the previous section improve the target performance.

We assume a virtual, fully connected system bidirectional links: 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, and both t_s and t_w are normalized to it.

The influence of optimization rules on performance depends on how the collective operations are implemented. We assume implementations which are mentioned in the literature as most widely used and which are available in our version of MPI. This enables a comparison of our estimates with experimental results.

All three collective operations involved in our optimization rules – broadcast, reduction and scan – are implementable using a butterfly-like communication pattern [5, 9]. It proceeds in $\log p$ phases, in which segments of length m are exchanged pairwise between processors. The only difference is in the computations: there are no computations in broadcast, reduction performs one base operation per element, and scan has two base operations:

$$T_{bcast} = \log p \cdot (t_s + m \cdot t_w) \quad (15)$$
$$T_{reduce} = \log p \cdot (t_s + m \cdot (t_w + 1)) \quad (16)$$
$$T_{scan} = \log p \cdot (t_s + m \cdot (t_w + 2)) \quad (17)$$

Let us illustrate our estimation technique on the SS2-Scan rule. 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, whose time we ignore. The major part is a scan on a list of pairs, with the base operator defined in the rule formulation, which requires time: $\log p \cdot (t_s + m \cdot (2 \cdot t_w + 6))$.

Therefore, the SS2-Scan optimization pays off iff:

$$t_s > 2 \cdot m$$

Remind that both sides are normalized w.r.t. the time of one computation. Thus, for the butterfly implementation of scan, rule SS2-Scan should be applied if the machine has a high start-up cost and/or the blocks in the processors are 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. If scan is implemented suboptimally on a virtual linear array with pipelining, then the SS2-rule worsens the performance, and thus must not be applied [6].

The performance estimates for our rules are summarized in Table 1. For each rule, we state the time before and the time after applying the rule. A factor of $\log p$ appears in all estimates; we omit it in the table entries. 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 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	$2t_s + m(2t_w + 3)$	$t_s + m(2t_w + 3)$	always
SR-Reduction	$2t_s + m(2t_w + 3)$	$t_s + m(2t_w + 4)$	$t_s > m$
SS2-Scan	$2t_s + m(2t_w + 4))$	$t_s + m(2t_w + 6)$	$t_s > 2m$
SS-Scan	$2t_s + m(2t_w + 4)$	$t_s + m(3t_w + 8)$	$t_s > m(t_w + 4)$
BS-Comcast	$2t_s + m(2t_w + 2)$	$t_s + m(t_w + 2)$	always
BSS2-Comcast	$3t_s + m(3t_w + 4)$	$t_s + m(t_w + 5)$	$t_w + \frac{1}{m} t_s > \frac{1}{2}$
BSS-Comcast	$3t_s + m(3t_w + 4)$	$t_s + m(t_w + 8)$	$t_w + \frac{1}{m} t_s > 2$
BR-Local	$2t_s + m(2t_w + 1)$	m	always
BSR2-Local	$3t_s + m(3t_w + 3)$	$3m$	always
BSR-Local	$3t_s + m(3t_w + 3)$	$4m$	$t_w + \frac{1}{m} \cdot t_s \geq \frac{1}{3}$

Table 1. Performance of optimizations

5. Case Study: Polynomial Evaluation

Let us present a simple case study to demonstrate the methodical use of our optimization rules. We do not claim the optimality of the obtained target program.

We consider the problem of evaluating a polynomial,

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

on m points: $y_1, \ldots, y_m$. The list of coefficients, as, is distributed so that processor i keeps value a_i, and the values $ys = [y_1, \ldots, y_m]$ are stored in the first processor.

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

$$Ev_in = bcast\,;\ scan\,(*)\,;\ map2\,(*)\,as\,;\ reduce\,(+) \quad (18)$$

Program (18) takes ys as input in the first processor and proceeds, stage-by-stage, as follows:

- $bcast$ broadcasts ys to all other processors;
- $scan\,(*)$ computes result $y^i = [y_1^i, \ldots, y_m^i]$ in each processor i, $i = 1, \ldots, n$;
- $map2$ is a map defined on two lists of equal size: in program (18), stage $map2\,(*)\,as$ computes in each processor, i, the following list: $[a_i * y_1^i, \ldots, a_i * y_m^i]$;
- finally, $reduce\,(+)$ sums up the obtained intermediate lists elementwise over the processors and puts the result list, $\left[\ \left(\ \sum_{i=1}^n a_i * y_1^i\ \right), \ldots, \left(\ \sum_{i=1}^n a_i * y_m^i\ \right)\ \right]$, into the first processor.

Program (18) contains three collective operations. The first two of them can be fused by rule BS-Comcast, for which Table 1 guarantees the improvement of performance.

The instantiation of rule BS-Comcast in our case is:

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

where operation op_poly is obtained from generic operator op_comp in the rule by substituting concrete operator, $*$:

$$op_poly\ k \stackrel{\text{def}}{=} pair\ ;\ repeat\,(e, o)\,k\ ;\ \pi_1$$
$$\text{where } e\,(t, u) = (t, u * u),\ o\,(t, u) = (t * u, u * u)$$

After applying BS-Comcast to (18), we get the program:

$$bcast\ ;\ map_{\#}\ op_poly\ ;\ map2\,(*)\,as\ ;\ reduce\,(+) \quad (19)$$

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

$$op_new\ k\ x\ y \stackrel{\text{def}}{=} (op_poly\ k\ x) * y$$

we can fuse them into one stage in the final program:

$$Ev_fin = bcast\,;\ map2_{\#}\,(op_new\ as)\,;\ reduce\,(+) \quad (20)$$

where $map2_{\#}$ is a $map_{\#}$ on two lists of equal size.

The experiments with rule BS-Comcast on a Parsytec 64-node machine with MPICH 1.0 are presented in Figure 7:

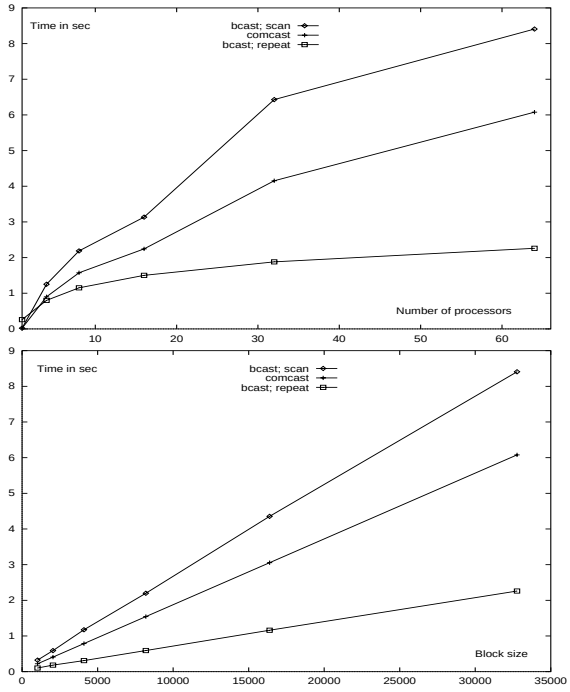

Figure 7. Run time improvement due to rule BS-Comcast

The upper plot shows the dependence of run time on the number of processors, for a fixed block size; the lower plot shows the dependence on the block size. The experiments confirm that the rule indeed improves the run time as indicated in Table 1.

The curves "bcast; scan" in both plots correspond to the left-hand side of the rule. For the right-hand side, we compare two possibilities discussed in Subsection 3.4: curve "comcast" corresponds to the cost-optimal version, and "bcast; repeat" corresponds to the more time-efficient version, demonstrated in Figure 6, which we used in all rules of group Comcast.

6. Discussion and Acknowledgements

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. We have sketched an optimization 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. The advantage of our formal approach is that the rules are independent of the way in which the collective operations are implemented on the particular machine. However, we take the implementation into account when estimating the target efficiency.

We can distinguish between two groups of rules. *General rules* refer to the traditional collective operations known from the literature. This group includes 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* require new collective operations like $reduce_bal$, $scan_bal$, etc. A rule from this group can be used only if the corresponding collective operation is implemented on a particular machine.

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 [16]. 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.

We have presented a very simple case study and only preliminary experimental results. We have another example which demonstrates the use of rule SR2-Reduction in the design of an asymptotically optimal algorithm [6].

We are grateful to Holger Bischof for his help in the computer experiments, to Peter Sanders for a fruitful discussion on the comcast pattern, and also to Marco Aldinucci, Christoph A. Herrmann and Susanna Pelagatti for helpful comments on the manuscript. The anonymous referees were helpful in improving the quality of presentation.

This work was supported by the travel grant from the German-Italian cooperation programme Vigoni.

References

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

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

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

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

[15] 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.

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