Towards Formally-Based Design of Message Passing Programs

Sergei Gorlatch

Abstract— **We present a systematic approach to the development of message passing programs. Our programming model is SPMD, with communications restricted to collective operations: scan, reduction, gather, etc. The design process in such an architecture-independent language is based on correctness-preserving transformation rules, provable in a formal functional framework. We develop a set of design rules for composition and decomposition: e.g., scan followed by reduction is replaced by a single reduction, and global reduction is decomposed into two faster operations. The impact of the design rules on the target performance is estimated analytically and tested in machine experiments. As a case study, we design two provably correct, efficient programs using the Message Passing Interface (MPI) for the famous maximum segment sum problem, starting from an intuitive but inefficient algorithm specification.**

Keywords— **Message passing, MPI, Collective operations, Transformations, Systematic program design, Reduction, Scan, Homomorhisms, Skeletons, Maximum segment sum.**

I. Introduction

One of the main challenges in the current parallel programming is the trade-off between the goals of performance, provable correctness and portability. To maximize performance, the programmer has to take into account many specific details of the parallel architecture. However, these details severely complicate the design of a provably correct target program. Also in order to ensure portability and, thus, the reuse of the programming effort, the machine-specific considerations should be avoided as much as possible, without sacrificing the performance.

The challenge has been addressed by the broad research on parallel models and languages:

- at the high-level end of the spectrum, there are formally-based approaches that use abstract parallel constructs for ensuring correctness and portability, e.g., the Bird-Meertens formalism (BMF) [1] and the transformational SAT approach based on it [2], skeletons [3], [4], and abstract parallel machines [5]. The current research concentrates on yet not well-understood issues, e.g.: how should abstract constructs be composed together to build a parallel program, and how can the most cost-effective design alternative be chosen [6].
- at the low-level end, there is a rich body of knowledge about parallel algorithms, practically-used design solutions and performance benchmarks. The current trend is towards standardization, with a notable example of the performance-oriented and still portable Message Passing Interface (MPI) [7], towards improving the performance predictability in the models like BSP [8] and LogP [9], and towards introducing higher-level features as in HPF. What is lacking at the low-level end is a sound methodology for designing algorithms and programs.

This brief overview (for the details of particular models and languages see an excellent survey [10]) shows that the both ends of the spectrum are actually moving towards each other. Such a development raises the need, and at the same time it creates the prerequisites, for finding a common ground, on which the advantages of both high-level and low-level models could be combined, thus enabling the design of correct, efficient and portable parallel programs.

In this paper, we argue that the SPMD model, with communications restricted to collective operations, allows to combine the advantages of both formally-based and performance-oriented approaches. Our main conribution is a design methodology, which has its formal roots in the functional Bird-Meertens formalism (BMF) and aims at designing efficient message passing programs with collective operations in MPI.

The features of our approach are as follows:

- *the formal base*, provided by BMF, enables reasoning about program properties, optimizing transformations, and implementation techniques; in particular, the parallelism of program pieces is captured via the formal notion of *homomorphism*;
- *the target programs* are expressed in MPI with exclusively collective operations (scan, reduction, gather, etc.) in the SPMD style; such programs are well-structured, they implement many practical algorithms efficiently [11], and are highly portable;
- *the design process* consists of two consecutive phases: 1) the algorithm design phase is architecture-independent: it starts from a correct but probably inefficient version/specification and proceeds by correctness-preserving transformations, aiming at mostly asymptotical complexity improvements; 2) the program design phase is implementation-oriented: it transforms the algorithm into an efficient representation for a particular target machine;
- *the design rules* govern algorithm and program development: they include "software-engineering" rules for composition (building a program of pieces), and decomposition (breaking big program pieces into smaller ones), as well as more complex rules;
- *the target performance* guides the design process: the impact of the design rules on performance is described analytically for different classes of parallel machines.

S. Gorlatch is with Universität Passau, D–94030 Passau, Germany. E-mail: gorlatch@fmi.uni-passau.de

The particular contributions and the structure of the paper are as follows:

– We demonstrate how MPI programs with collective operations can be modeled in BMF as compositions of (possibly adjusted) homomorphic functions (Section II).

– We present two composition rules in BMF: the first rule transforms a sequence of scan and reduction into a single reduction, the second rule transforms a sequence of two scans into a single scan. In the MPI setting, these rules fuse two global communications into one, which often improves the target performance (Section III).

– We demonstrate our design approach with a case study: starting from an intuitively clear but very inefficient (cubic time complexity) algorithm for the maximum segment sum problem, we arrive by transformation at two target programs of logarithmic complexity (Section IV).

– We describe in BMF a generic, hypercube-like implementation for homomorphisms, and adapt it for the MPI collective operations (Section V).

– We present decomposition rules which split one collective operation into a composition of two operations, with a potentially better performance, and consider a possibility of combining composition and decomposition in a single design rule (Section VI).

– We study the impact of the developed design rules on the target program performance by analytical estimates in diverse parallel models: hypercube, BSP, and linear pipeline, as well as by experiments with the target MPI programs on parallel machines (Section VII).

II. Modeling MPI Programs in BMF

Message passing programs are usually written in the SPMD (Single Program Multiple Data) style, with the processor program being a sequence of computation stages and communication stages.

We will use a simplified pseudocode for representing message passing programs, with computations represented by function calls, and communications represented by calls to the MPI collective operations. For example, the following program consists of a computation stage, `f`, followed by two collective communication stages, scan and reduction:

```
Program Example (my_pid, ...);
  f (...);
  MPI_Scan (⊗);                    (1)
  MPI_Reduce (⊕);
```

As usual in the SPMD style, program `Example` runs on all processors of a machine and refers to the logical number of a processor, `my_pid`, as parameter. An ellipsis, `...` , marks the omission of minor details.

We assume for simplicity that the output of each stage is consumed by the next stage, so that no intermediate variables are necessary.

A. From MPI to BMF

To enable formal reasoning about message passing programs, we represent them in the functional framework of the Bird-Meertens formalism (BMF) [1]. For simplicity of presentation, our MPI programs will be defined over one-dimensional arrays which are modeled in BMF by non-empty, finite lists.

The computation stage of (1), denoted by `f (...)`, means that every processor computes function `f` on its local data, independently of the other processors. In BMF, this can be modeled by the functional *map*, which applies unary function f to all elements of a list:

$$map\ f\ [x_1, x_2, \dots, x_n] \ = \ [\, f\, x_1,\ f\, x_2,\ \dots,\ f\, x_n\,] \tag{2}$$

Here and in the following, function application is denoted by juxtaposition, has the highest binding power and associates to the left. Note that function f in (2) may be arbitrarily complex, including for instance conditional branching, so that computations in a processor depend on its current data and on the processor's id. This allows us to model not only purely data-parallel programs but also some irregular and task-parallel aspects.

Collective operation `MPI_Reduce` in (1) combines data in the processors, with an associative binary operator, which can either be defined by the user, or be predefined, e.g., addition or maximum. We capture this genericity by using parameter operation $\oplus$ in `MPI_Reduce (`$\oplus$`)`; other arguments of MPI calls will be usually omitted for brevity. In BMF, reduction is expressed by the functional *red*, which expects an associative operator, $\oplus$, and a list:

$$red\,(\oplus)\,[x_1, x_2, \dots, x_n] \ = \ x_1 \oplus x_2 \oplus \dots \oplus x_n \tag{3}$$

Another important collective operation is `MPI_Scan(`$\otimes$`)`, which performs the parallel scan calculation: on each processor, the result is an accumulation of the data from processors with either smaller or bigger ranks, using associative operator $\otimes$. Corresponding two versions of scan, prefix and suffix, are denoted by $pref$ and suf, respectively:

$$pref\,(\otimes)\,[x_1, x_2, \dots, x_n] \ = \ [x_1,\ x_1 \otimes x_2,\ \dots,\ x_1 \otimes x_2 \otimes \dots \otimes x_n] \tag{4}$$

$$suf\,(\otimes)\,[x_1, x_2, \dots, x_n] \ = \ [x_1 \otimes x_2 \otimes \dots \otimes x_n,\ \dots,\ x_{n-1} \otimes x_n,\ x_n] \tag{5}$$

The structure of MPI programs is a sequential composition of parallel stages: computations and collective operations. We model this by functional composition, $\circ$, defined as follows:

$$(f \circ g)\, x \ \stackrel{\text{def}}{=} \ f\,(g\, x) \tag{6}$$

Summarizing, the function computed by MPI program (1) is expressed in BMF as follows:

$$Example \ = \ red\,(\oplus) \circ pref\,(\otimes) \circ map\, f \tag{7}$$

Note that the composed functions in (7) are computed from right to left: *map*, then $pref$, etc.

B. Parallelism and Transformations in BMF

The parallelism of program stages is described in BMF using the notion of homomorphism.

Definition (homomorphism [12]) Function h on lists is a *homomorphism* iff there exists operator $\circledast$ such that, for arbitrary lists x and y, and list concatenation $+\!\!+$:

$$h\,(x +\!\!+\, y) \;=\; h\,x \;\circledast\; h\,y \qquad (8)$$

□

In words: the value of h on a concatenated list can be computed by applying $\circledast$ to the values of h on the pieces of the list, see Figure 1. Since the computations of $h\,x$ and $h\,y$ are independent of each other, they can be performed in parallel.

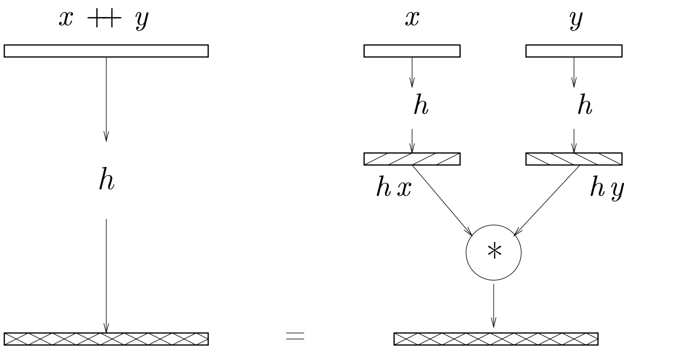

Fig. 1. Computation of homomorphism h according to equality (8): sequentially (left) and in parallel (right).

Simple examples of homomorphisms are the basic functions of BMF: map, reduction and scan, which we have used in (7) to model MPI program (1). Although the notion of homomorphism seems to be quite restrictive, many practically relevant functions have been shown to be adjustable to the homomorphic format by introducing auxiliary function(s), i.e. at the price of extra computations [2], [13].

Homomorphisms can be computed in parallel by partitioning the input list into an arbitrary concatenation of segments (blocks), as expressed by the *promotion property*:

$$h \;\circ\; red\,(+\!\!+) \;=\; red\,(\circledast) \;\circ\; map\,h \qquad (9)$$

Programs in BMF, like (7), can be transformed by applying, step-by-step, semantics-preserving equational rules, such as (9). We will use the following notation for a stepwise derivation: it starts with *Expression.initial*, transforms at each step a current expression into a semantically equivalent form by applying one or several transformation rules, and leads to the resulting *Expression.final*:

$$\begin{array}{ll} Expression.initial & \{\text{Rule(s) applied}\} \\ \ldots \;\; = & \{\text{More rules applied}\} \\ Expression.final & \end{array}$$

Homomorphisms are, thus, well-parallelizable functions, amenable to formal transformations and related to the collective operations of MPI. We will use homomorphisms for modeling individual MPI program stages. Let us now look at how programs are constructed of such stages.

III. Composition Rules

Complex parallel programs are usually "engineered" by composing simpler parallel stages. Sequential composition, expressed by ; in MPI and by $\circ$ in BMF, plays a key rôle in the program construction: it reflects data and/or control dependences between different pieces of a program. We will pay special attention to the compositions of collective operations because they may be quite expensive, especially if processors synchronize their activity in the points of composition. Our goal is to find out, when such compositions can be optimized.

We will present two composition optimization rules, whose proofs (see [14]) proceed by adjustment to the homomorphic format (8). The rules exploit specific properties of the operators involved in reduction and scan. We say that operator $\otimes$ *distributes forwards over* $\oplus$ iff:

$$a \otimes (b \oplus c) \;=\; (a \otimes b) \oplus (a \otimes c) \qquad (10)$$

Similarly, $\otimes$ *distributes backwards over* $\oplus$ iff:

$$(b \oplus c) \otimes a \;=\; (b \otimes a) \oplus (c \otimes a) \qquad (11)$$

Theorem 1 (Scan-Reduce Composition) For two binary, associative operators $\oplus$ and $\otimes$:

- if $\otimes$ distributes forwards over $\oplus$, i.e. (10) holds, then
$$red(\oplus) \circ pref(\otimes) \;=\; \pi_1 \;\circ\; red\,(\langle\!\langle \oplus, \otimes \rangle\!\rangle) \;\circ\; map\,pair \qquad (12)$$
- if $\otimes$ distributes backwards over $\oplus$, i.e. (11) holds, then
$$red(\oplus) \circ suf(\otimes) \;=\; \pi_1 \circ red\,(<\!\oplus, \otimes\!>) \;\circ\; map\,pair \qquad (13)$$

□

The functions used in (12)–(13) are defined as follows:

$$pair\,a \;\stackrel{\text{def}}{=}\; (a, a), \qquad (14)$$

$$\pi_1\,(a, b) \;\stackrel{\text{def}}{=}\; a \qquad (15)$$

$$(s_1, r_1)\,\langle \oplus, \otimes \rangle\,(s_2, r_2) \;\stackrel{\text{def}}{=}\; (s_1 \oplus (r_1 \otimes s_2)\,, r_1 \otimes r_2) \qquad (16)$$

$$(s_1, r_1)\,<\!\oplus, \otimes\!>\,(s_2, r_2) \;\stackrel{\text{def}}{=}\; ((s_1 \otimes r_2) \oplus s_2\,, r_1 \otimes r_2) \qquad (17)$$

In words: *pair* is the function which creates a pair of an element, π_1 is a projection yielding the first element of a pair, and $\langle \oplus, \otimes \rangle$ and $<\!\oplus, \otimes\!>$ are new operators defined via $\oplus$ and $\otimes$ and working on pairs.

For MPI programs, Theorem 1 provides the following transformation:

```
[ MPI_Scan( ⊗ );          [ Make_pair;
[ MPI_Reduce( ⊕ );   ⟹    [ MPI_Reduce(⟨⟨⊕,⊗⟩⟩);
                           [ if my_pid==ROOT
                           [   then Take_first;
```

Here, functions `Make_pair` and `Take_first` implement *map pair* and π_1 of (12), respectively. The result of reduction in MPI is a single datum stored in the processor with `my_pid = ROOT`. Note that the scan-reduce transformation is applicable directly to our example program (1); *it fuses two collective operations into one*, with simple local computations beforehand and afterwards.

Our next design rule transforms a composition of two scans (see [14] for a proof):

Theorem 2 (Scan-Scan Composition) For arbitrary associative operators $\oplus$ and $\otimes$:

- if $\otimes$ distributes forwards over $\oplus$, then

$$pref\,(\oplus) \circ pref\,(\otimes) = map\,\pi_1 \circ pref\,(\langle\oplus,\otimes\rangle) \circ map\,pair \tag{18}$$

- if $\otimes$ distributes backwards over $\oplus$, then

$$suf\,(\oplus) \circ suf\,(\otimes) = map\,\pi_1 \circ suf\,(<\oplus,\otimes>) \circ map\,pair \tag{19}$$

where $pair$, π_1, $\langle\oplus,\otimes\rangle$, $<\oplus,\otimes>$ are defined by (14)–(17). □

The theorem has the following MPI formulation:

$$\begin{bmatrix}\texttt{MPI\_Scan(}\otimes\texttt{);}\\ \texttt{MPI\_Scan(}\oplus\texttt{);}\end{bmatrix} \Longrightarrow \begin{bmatrix}\texttt{Make\_pair;}\\ \texttt{MPI\_Scan(}\langle\oplus,\otimes\rangle\texttt{);}\\ \texttt{Take\_first;}\end{bmatrix}$$

where functions `Make_pair` and `Take_first` are as explained for the previous theorem.

The composition rules given by Theorems 1 and 2 have the following important properties: (1) their correctness is proved formally; (2) they are parameterized by $\oplus$ and $\otimes$, and thus are valid for a variety of particular base operators; (3) they can be applied independently of the architecture of the target parallel machine; (4) they are valid for all possible implementations of the involved collective operations.

IV. Algorithm Design: Case Study of MSS

In this section, we demonstrate how the composition rules provided by Theorem 1 and Theorem 2 can be exploited in the design of parallel algorithms. The idea is to start with an intuitive, obviously correct but probably inefficient version of an algorithm and proceed by applying semantically sound transformation rules, until an efficient algorithm is obtained.

We consider the famous *Maximum Segment Sum* (MSS) problem – a *programming pearl* [15], studied by many authors [1], [12], [13], [16], [17]. Given a list of numbers, function *mss* finds a contiguous list segment, whose members have the largest sum among all such segments and returns this sum. For example:

$$mss\,[\,2,-4,2,-1,6,-3\,] = 7$$

where the result is contributed by the segment $[2,-1,6]$.

Let us start with the following intuitive, obviously correct specification of *mss*:

$$mss = red\,(\uparrow) \circ map\,(red\,(+)) \circ segs \tag{20}$$

where (20) consists of three stages, from right to left:

- *segs* : yields the list of all segments of the original list;
- $map\,(red\,(+))$: for each segment, computes the sum of its elements by the reduction with the usual addition, $+$; the list of segment sums is thus obtained;
- $red\,(\uparrow)$: the maximum segment sum is computed by the reduction with operator $\uparrow$ that computes the maximum of its two arguments.

Let us introduce functions which yield all initial and all closing segments of a list:

$$\begin{aligned} inits\,[x_1,x_2,\ldots,x_n] &= [[x_1],[x_1,x_2],\ldots,[x_1,x_2,\ldots,x_n]]\\ tails\,[x_1,x_2,\ldots,x_n] &= [[x_1,x_2,\ldots,x_n],\ldots,[x_{n-1},x_n],[x_n]] \end{aligned}$$

Now, function *segs* can be specified as follows:

$$segs = red\,(+\!\!+) \circ map\,tails \circ inits$$

Substituting this into (20), we obtain our first algorithm for computing *mss*:

$$mss = red\,(\uparrow) \circ map\,(red\,(+)) \circ red\,(+\!\!+) \circ map\,tails \circ inits \tag{21}$$

Algorithm (21) is obviously correct but, for example in the cost model of [1], it has a poor time complexity – $O\,(n^3)$ for an input list of length n, in both sequential and parallel cases, due to the communication costs of stage $red\,(+\!\!+)$.

To design a better algorithm, we will start from (21) and make use of the rules provided by Theorem 1 and also of some standard BMF equalities [1]:

$$\begin{aligned} map\,(f \circ g) &= map\,f \circ map\,g && (22)\\ map\,f \circ red\,(+\!\!+) &= red\,(+\!\!+) \circ map\,(map\,f) && (23)\\ red\,(\oplus) \circ red\,(+\!\!+) &= red\,(\oplus) \circ map\,(red\,(\oplus)) && (24)\\ pref\,(\oplus) &= map\,(red\,(\oplus)) \circ inits && (25)\\ suf\,(\oplus) &= map\,(red\,(\oplus)) \circ tails && (26)\\ inits \circ map\,f &= map\,(map\,f) \circ inits && (27) \end{aligned}$$

Note that (23) and (24) are just special cases of(9), since both $map\,f$ and $red\,(\oplus)$ are homomorphisms.

The derivation shown in Figure 2 exploits the associativity of $\uparrow$ and $+$, and the fact that $+$ distributes over $\uparrow$ backwards: $(a \uparrow b) + c = (a + c) \uparrow (b + c)$. Therefore, we use Theorem 1 for suffix, with $\otimes = +$ and $\oplus = \uparrow$, in the sequence of design steps which starts from (21).

The derivation in Figure 2 arrives at the underlined expression for *mss*; its time complexity is $O\,(n)$ sequentially and $O\,(\log n)$ in parallel, which is a serious improvement over the cubic complexity of the initial algorithm (21).

The corresponding MPI program reads as follows:

```
Program MSS-1(my_pid, ...);
  Make_pair;
  MPI_Scan ( <↑,+ > );                     (28)
  Take_first;
  MPI_Reduce ( ↑ );
```

$$\begin{aligned}
mss &= red\,(\uparrow) \circ map\,(red\,(+)) \circ red\,(+\!\!+) \circ map\,tails \circ inits && \{\text{Eq. (23), (24)}\}\\
&= red\,(\uparrow) \circ map\,(red\,(\uparrow)) \circ map\,(map\,(red\,(+))) \circ map\,tails \circ inits && \{\text{Eq. (22), (26)}\}\\
&= red\,(\uparrow) \circ map\ \left(\, red\,(\uparrow) \circ suf\,(+)\,\right) \circ inits && \{\text{Th. 1: Eq. (13)}\}\\
&= red\,(\uparrow) \circ map\ \left(\,\pi_1 \circ red\left(<\uparrow,+>\right) \circ map\,pair\,\right) \circ inits && \{\text{Eq. (22), (27)}\}\\
&= red\,(\uparrow) \circ map\,\pi_1 \circ map\,(red\left(<\uparrow,+>\right)) \circ inits \circ map\,pair && \{\text{Eq. (25)}\}\\
&= \underline{red\,(\uparrow) \circ map\,\pi_1 \circ pref\left(<\uparrow,+>\right) \circ map\,pair} && \{\text{Eq. (29)}\}\\
&= \pi_1 \circ red\,(\Uparrow) \circ pref\left(<\uparrow,+>\right) \circ map\,pair && \{\text{Th.1: Eq. (12) , and (22)}\}\\
&= \pi_1 \circ \pi_1 \circ red\left(\langle \Uparrow, <\uparrow,+> \rangle\right) \circ map\,(pair \circ pair)
\end{aligned}$$

Fig. 2. Formal program derivation for MSS in the BMF notation

Program (28) still has two collective communications, and that suggests as the next design step to try and merge those communications together:

- Extend $\uparrow$ to pairs: $(a, b) \Uparrow (c, d) \stackrel{\text{def}}{=} (a \uparrow c\,, b \uparrow d)$, and note that $\Uparrow$ is associative and also that:

$$red\,(\uparrow) \circ map\,\pi_1 = \pi_1 \circ red\,(\Uparrow) \tag{29}$$

- Note that $<\uparrow,+>$ distributes forwards over $\Uparrow$, which allows to use Theorem 1 for prefix, with $\otimes = <\uparrow,+>$ and $\oplus = \Uparrow$.

The derivation continues from the underlined expression as shown in Figure 2. Denoting $\pi_1^2 \stackrel{\text{def}}{=} \pi_1 \circ \pi_1$ and $pair^2 \stackrel{\text{def}}{=} pair \circ pair$, we obtain the following target program:

$$mss = \pi_1^2 \circ red\left(\langle \Uparrow, <\uparrow,+> \rangle\right) \circ map\,(pair^2) \tag{30}$$

The semantics of operator $\langle \Uparrow, <\uparrow,+> \rangle$ which works on pairs of pairs (quadruples) can be expressed in terms of $\uparrow$ and $+$, if we recall the definitions of $\langle\,\rangle$, $<>$ and $\Uparrow$:

$$\begin{aligned}
&((r_1, s_1), (t_1, u_1))\ \langle \Uparrow, <\uparrow,+> \rangle\ ((r_2, s_2), (t_2, u_2)) \stackrel{\text{def}}{=}\\
&\quad ((r_1 \uparrow r_2 \uparrow (t_1 + s_2)\,, s_1 \uparrow (u_1 + s_2))\,, (t_2 \uparrow (t_1 + u_2)\,, u_1 + u_2))
\end{aligned}$$

Note that the elements of a quadruple in the above definition – r, s, t, u – all have an intuitive meaning: *mss* itself, maximum sum of an initial segment, etc. Unlike other derivations for *mss* [13], our approach proceeds without using problem-specific knowledge.

The MPI version of target program (30) is as follows, with the creation and projection functions for quadruples:

```
Program MSS-2 (my-pid, ...);
  Make_quadruple;                                   (31)
  MPI_Reduce (⟨⇑,<↑,+>⟩);
  if my_pid==ROOT then Take_first;
```

We have thus obtained two MPI programs for the MSS problem: (28) and (31). Both programs have logarithmic time complexity, so to compare their performance and, in general, to assess the influence of design rules on the target performance we will turn now from our architecture-independent view towards implementation considerations.

V. Implementation: Hypercube Schema

In this section, we extend our formal base, in order to reason formally not only about programs constructed of collective operations, but also about their implementation and performance.

We choose the hypercube as our basic parallel implementation model because, as often argued in the literature (for instance, [18, p. 51]), many hypercube algorithms are as fast asymptotically as the optimal PRAM algorithms, and they can be translated systematically to other parallel topologies. Furthermore, hypercube-like algorithms are very widely used in the contemporary implementations of the MPI collective operations.

Parallel implementations on a hypercube can be described formally using function *swap*, which captures the hypercube communication pattern: $swap\ d\,(\oplus, \otimes)$ is computed by pairwise, bidirectional communication in dimension d, followed by the application of $\oplus$ to the exchanged data in processors with lower ranks and the application of $\otimes$ in processors with higher ranks. Swap has an obvious SPMD implementation using, e.g., primitives `MPI_Send` and `MPI_Recv` [19].

We abbreviate multiple compositions (loops) of *swap*s:

$$\bigcirc_{d=1}^{k} swap\ d\,(\oplus, \otimes) \stackrel{\text{def}}{=} swap\ k\,(\oplus, \otimes) \circ \ldots \circ swap\ 1\,(\oplus, \otimes)$$

Dimensions of the hypercube can be "swapped" also in the decreasing order, which we denote by $\bigcirc_{d=k}^{1}(swap\,d\,(\oplus, \otimes))$. Figure 3 illustrates a sequence of swaps on a two-dimensional hypercube with 4 nodes: solid arrows show communications, dashed lines show idle links.

To obtain a hypercube implementation for a particular function, we must customize operators $\oplus$ and $\otimes$ in the swaps. A general approach for that, based on the notion of distributable homomorphism, is presented in [2].

As a simple example, let us consider the so-called *global reduction*, informally defined as follows: $allred\,(\odot)\,x = [red\,(\odot)\,x, \ldots, red\,(\odot)\,x]$, where $red\,(\odot)$ is from (3).

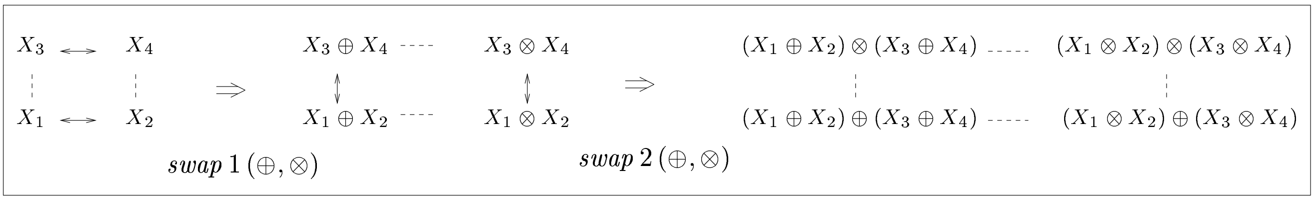

Fig. 3. Composition of two swaps on a virtual hypercube

A hypercube implementation for the global reduction is the following composition of swaps:

$$\mathit{allred}\,(\odot) \;=\; \mathop{\bigcirc}_{d=1}^{k} (\mathit{swap}\; d\,(\odot, \odot)) \tag{32}$$

Note that this simple implementation of reduction as a sequence of swaps on the hypercube is the algorithm usually presented in the literature [18].

The MPI global reduction operation, `MPI-Allreduce`, presumes that there is a block (segment) of equal length in each processor. We capture this by the specific, *segmented* version of global reduction: function *allred-seg* accepts a list of sublists (segments) of equal length, say m, and applies operator $\odot$ to the segments elementwise. For example, segmented global reduction with usual addition $+$, for $m = 4, p = 4$, works as follows:

$$\begin{aligned} &\mathit{allred\text{-}seg}\,(+)\;[\,[x_1, y_1, z_1, u_1], [x_2, y_2, z_2, u_2],\\ &\qquad [x_3, y_3, z_3, u_3], [x_4, y_4, z_4, u_4]\,] \;=\\ &\big[\,[X,Y,Z,U],\, [X,Y,Z,U],\, [X,Y,Z,U],\, [X,Y,Z,U]\,\big] \end{aligned} \tag{33}$$

with $X{=}\sum_{i=1}^{4} x_i, Y{=}\sum_{i=1}^{4} y_i, Z{=}\sum_{i=1}^{4} z_i, U{=}\sum_{i=1}^{4} u_i$.

A hypercube implementation for *allred-seg* and, thus, also for `MPI_Allreduce`, is obtained by specializing (32) with function *zip* for elementwise computations, where $\mathit{zip}(\odot)\,([x_1, \dots, x_n], [y_1, \dots, y_n]) \stackrel{\mathrm{def}}{=} [x_1{\odot}y_1, \dots, x_n{\odot}y_n]$:

$$\mathit{allred\text{-}seg}\,(\odot) \;=\; \mathop{\bigcirc}_{d=1}^{k} (\mathit{swap}\; d\,(\mathit{zip}(\odot), \mathit{zip}(\odot))) \tag{34}$$

Similarly, we model operation `MPI_Scan` by function $\mathit{pref\text{-}seg}(\odot)$ which, as shown in [2], has the following hypercube implementation:

$$\begin{aligned} \mathit{pref\text{-}seg}\,(\odot) \;&=\; \mathit{map}^2\, \pi_1 \;\circ\\ &\mathop{\bigcirc}_{d=1}^{k} (\mathit{swap}\; d\,(\mathit{zip}(\oplus), \mathit{zip}(\otimes))) \;\circ\; \mathit{map}^2\; \mathit{pair} \end{aligned} \tag{35}$$

where $\mathit{map}^2 f \stackrel{\mathrm{def}}{=} \mathit{map}\,(\mathit{map}\, f)$, function *pair* is defined by (14), and $\oplus$ and $\otimes$ are as follows:

$$\begin{aligned} (s_1, r_1) \oplus (s_2, r_2) \;&=\; (s_1\,,\, r_1 \odot r_2)\\ (s_1, r_1) \otimes (s_2, r_2) \;&=\; (r_1 \odot s_2\,,\, r_1 \odot r_2) \end{aligned} \tag{36}$$

Hypercube algorithms: (34) for `MPI_Allreduce` and (35) for `MPI_Scan`, are most widely used in current MPI implementations. In the next section, we study their possible improvement.

VI. Decomposition and Combined Rules

Let us look more closely at program (34) for a segmented global reduction. As example (33) shows, the same result segments are computed on all processors, so that implementation (34) performs many redundant computations: even if we ignore the communication costs, its parallel speed-up is not better than $p/\log p$ on p processors [20].

Thus, as a monolithic collective operation, reduction may have suboptimal performance. Let us try to improve it by means of *decomposition*, i.e. by transforming one collective operation into a composition of two operations that avoid the mentioned redundancy.

A. Decomposition of Reduction

We consider `MPI_Allreduce` on p processors, with a segment of length m in each processor, where both m and p are powers of 2, and $p \leq m$. The idea of decomposition is to compute reduction in two steps: first, compute different parts of length m/p of the result in different processors and, second, combine them in each processor.

Let us introduce function *allred-parts* (for *global reduction-in-parts*) which performs the first step. We illustrate it on a list of two sublists, each with four elements, i.e., $m = 4, p = 2$:

$$\begin{aligned} &\mathit{allred\text{-}parts}\,(+)\;[\,[x_1, y_1, z_1, u_1], [x_2, y_2, z_2, u_2]\,] \;=\\ &\qquad [\,[x_1 + x_2\,,\, y_1 + y_2]\,,\, [z_1 + z_2\,,\, u_1 + u_2]\,] \end{aligned}$$

This function can be expressed by a restricted version of `MPI_Reduce_scatter` which we call `Red-scatter'`.

After computing $\mathit{allred\text{-}parts}\,(\odot)$, each processor keeps a $(1/p)$-th part of the result list, so the second step must combine those parts in each processor. This can be done with `MPI_Allgather`. We model it by function *allgather*: for a list of segments, it yields the list whose elements are all equal to the concatenation of the segments, e.g.:

$$\mathit{allgather}\,[[x_1, x_2], [x_3, x_4]] \;=\; [[x_1, x_2, x_3, x_4], [x_1, x_2, x_3, x_4]]$$

The envisioned two-step computation is expressed by the following *reduction decomposition rule*:

$$\mathit{allred\text{-}seg}\,(\odot) \;=\; \mathit{allgather} \,\circ\, \mathit{allred\text{-}parts}\,(\odot) \tag{37}$$

or, in the MPI notation,

$$\texttt{MPI\_Allreduce}(\odot) \Longrightarrow \left[\begin{array}{l} \texttt{Red-scatter'}(\odot)\texttt{;}\\ \texttt{MPI\_Allgather;} \end{array}\right.$$

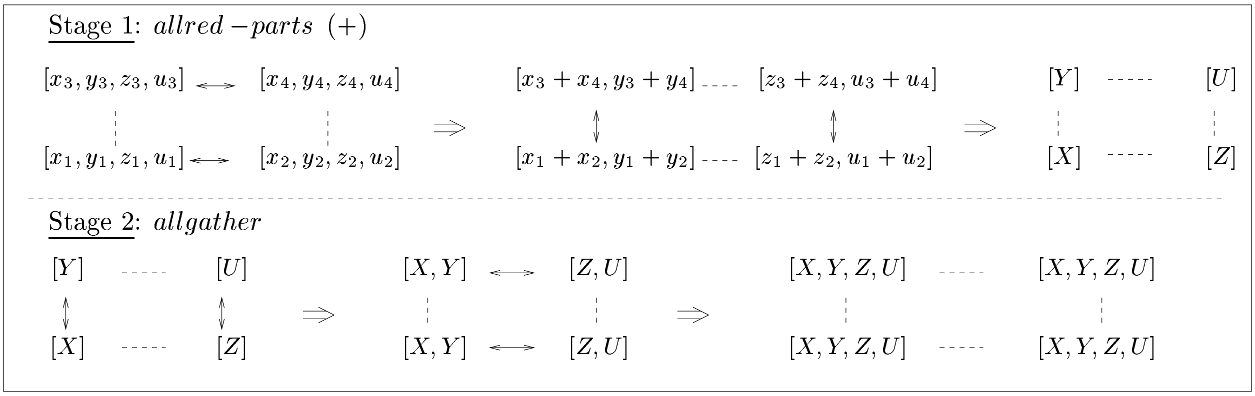

Fig. 4. Decomposed hypercube reduction: $X = \sum_{i=1}^{4} x_i, Y = \sum_{i=1}^{4} y_i, Z = \sum_{i=1}^{4} z_i, U = \sum_{i=1}^{4} u_i$.

Two more examples of decomposition rules for MPI, which we present without a proof, are as follows:

$$\texttt{MPI\_Bcast} \Longrightarrow \begin{bmatrix} \texttt{MPI\_Scatter;} \\ \texttt{MPI\_Allgather;} \end{bmatrix}$$

$$\texttt{MPI\_Reduce}(\odot) \Longrightarrow \begin{bmatrix} \texttt{Red-scatter'}(\odot)\texttt{;} \\ \texttt{MPI\_Gather;} \end{bmatrix}$$

In the next subsection, we deal with the implementation of decomposition rule (37).

B. Implementation of the Reduction Decomposition

To apply our hypercube implementation schema of form $\bigcirc_{d=1}^{k} swap\ d\,(\oplus, \otimes)$ to the decomposed reduction, we must customize swap-operators for both functions on the right-hand side of (37). Whereas list concatenation, $+\!\!+$, is the clear solution for $allgather$, function *allred-parts* is a more complex case. Let functions $left$ and $right$ yield the left and the right halves of a list, respectively. The customizing operators for *allred-parts* are illustrated by:

$$\begin{aligned} (left \circ zip\,(+))\,([x_1, y_1, z_1, u_1], [x_2, y_2, z_2, u_2]) &= \\ &[x_1 + x_2\ ,\ y_1 + y_2] \\ (right \circ zip\,(+))\,([x_1, y_1, z_1, u_1], [x_2, y_2, z_2, u_2]) &= \\ &[z_1 + z_2\ ,\ u_1 + u_2] \end{aligned}$$

The hypercube implementation of the decomposed reduction reads as follows:

$$\begin{aligned} allred\text{-}seg\,(\odot) \quad = \quad & \bigcirc_{d=k}^{1} (swap\,d\,(+\!\!+, +\!\!+)) \ \circ \qquad (38) \\ & \bigcirc_{d=1}^{k} (swap\,d\,(left \circ zip(\odot)\ ,\ right \circ zip(\odot))) \end{aligned}$$

Figure 4 illustrates how program (38) computes reduction $allred\text{-}seg\,(+)$, decomposed by (37), on the hypercube of four processors, each of which keeps a block of four elements. Each of the two program stages consists of two swaps, with the increasing and decreasing order of dimensions, respectively.

Note that, in spite of function *allred-parts* being semantically similar to operation `MPI_Reduce_scatter`, the actual implementations of the two are different. The general MPI call is usually implemented as reduction followed by scatter, which clearly destroys all potential advantages of the decomposition. We use a specific collective operation, `Red-scatter'`, which is implemented as illustrated by the first stage in Figure 4.

C. Combined Rule

Composition and decomposition rules can be sometimes combined. E.g., the composition of a segmented scan and a reduction can be transformed by a version of (12):

$$\begin{aligned} allred\text{-}seg\,(\oplus) \ \circ\ pref\text{-}seg\,(\otimes) \quad = \qquad\qquad (39) \\ map\,\pi_1 \ \circ\ allred\text{-}seg\,(\langle\!\langle \oplus, \otimes \rangle\!\rangle) \ \circ\ map^2\,pair \end{aligned}$$

Now, we can decompose the result reduction on the right-hand side by (37), arriving at the following expression:

$$map\,\pi_1 \ \circ\ allgather \circ\ allred\text{-}parts\,(\langle\!\langle \oplus, \otimes \rangle\!\rangle) \ \circ\ map^2\,pair$$

Since the last stage, projection, excludes a half of results, we can save communications in the gathering stage by applying the following rule

$$map\,\pi_1 \ \circ\ allgather \ = \ allgather \ \circ\ map^2\,\pi_1 \qquad (40)$$

The described application of three rules in sequence, (39), (37) and (40), can be viewed as a single design rule:

$$\begin{aligned} allred\text{-}seg\,(\oplus) \ \circ\ pref\text{-}seg\,(\otimes) \quad = \quad allgather\ \circ \qquad (41) \\ map^2\,\pi_1 \ \circ\ allred\text{-}parts\,(\langle\!\langle \oplus, \otimes \rangle\!\rangle) \ \circ\ map^2\,pair \end{aligned}$$

We call (41) the *combined rule*, with the following MPI representation:

$$\begin{bmatrix} \texttt{MPI\_Scan}(\otimes)\texttt{;} \\ \texttt{MPI\_Allreduce}(\oplus)\texttt{;} \end{bmatrix} \Longrightarrow \begin{bmatrix} \texttt{Map Make\_pair;} \\ \texttt{Red-scatter'}(\langle\!\langle \oplus, \otimes \rangle\!\rangle)\texttt{;} \\ \texttt{Map Take\_first;} \\ \texttt{MPI\_Allgather;} \end{bmatrix}$$

Here, `Map` applies a function to the block kept in a processor (this local computation will be usually implemented sequentially). The combined rule suggests, in addition to (39), one more possible transformation of both example program (1), and intermediate program (28) of the MSS case study. To choose the right rule to apply at a particular point of the design process, we need to study the impact of the design rules on the target program performance.

VII. Performance Estimation and Machine Experiments

In this section, we estimate the performance impact of the design rules in the hypercube and in other parallel models, and report our experimental results. We study collective operations in their segmented versions, in order to assess how the block size influences the performance.

A. Hypercube Performance: Basic Estimates

Our hypercube model has following properties. All base operators are assumed to take one unit of time. Communication links are bidirectional: two neighboring processors in one dimension of the hypercube can send blocks of size m to each other simultaneously in time $t_s + m \cdot t_w$, where t_s is the start-up time and t_w is the per-word transfer time. A processor is allowed to send/receive data on only one of its links at a time. We ignore the computation time it takes to split or concatenate segments within a processor.

As an example, let us estimate the time complexity of scan, implemented by (35). We ignore the costs of pairing and projecting, since they form just a small additive constant. There are $\log p$ swaps, with m elements communicated and at most two operations per element performed in each processor according to (36). This yields a time of

$$\log p \cdot (t_s + m \cdot (t_w + 2)) \tag{42}$$

Similarly, the time required by the global reduction implemented by program (34) is:

$$\log p \cdot (t_s + m \cdot (t_w + 1)) \tag{43}$$

B. Composition Rules on Hypercube

We start with the performance evaluation of the scan-scan composition rule. The time complexity of the left-hand side of rule (18) is, according to (42):

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

The right-hand side performs pairing at the beginning and projection at the end, whose time we ignore. Its major part is a usual scan but, this time, on a list of pairs, with the base operator defined by (16). Thus, the right-hand side of rule (18) requires a time of:

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

Therefore, optimization (18) pays off iff:

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

Thus, in the hypercube model, rules (18) and (19) should be applied if the machine has relatively high start-up costs and/or if the length of the blocks held in the processors is small. This result is in line with intuition: the scan-scan composition rules trade synchronization costs, expressed by the start-up, for additional computations which grow with the block length. Note that for big blocks the scan-scan composition rules may become impractical because of the additional memory consumption.

Let us now consider the scan-reduction composition rules. The time of the left-hand side of rule (39) follows from (42) and (43):

$$\log p \cdot (2 \cdot t_s + m \cdot (2 \cdot t_w + 3))$$

while the time of the right-hand side is as follows:

$$\log p \cdot (t_s + m \cdot (2 \cdot t_w + 3))$$

These estimates show that the scan-reduction composition rules on a hypercube save the start-up costs, without adding any new computation or data transfer costs, and without additional memory requirements. Thus, if one of these rules is applicable, then it will always bring a performance improvement.

C. Composition in BSP and on Linear Pipeline

The estimates of the previous subsections apply to the hypercube-like implementations of scan and reduction. Other implementations may exemplify completely different communication patterns and, thus, have to be studied separately.

Composition in BSP. Let us assess the scan-scan transformation in the BSP model. As shown in [21], the optimal, "transpose scan" algorithm has following BSP costs:

$$2 \cdot (m \cdot g + l) + m$$

where l is the cost of the barrier synchronization, and g is the single-word delivery cost, both normalized by the instruction rate of the processors. Then optimization (18) pays off if:

$$2 \cdot l \; > \; m$$

which, interestingly enough, is quite similar to the estimate obtained in Subsection VII-B for the hypercube algorithm. In [21], the implementers of BSP report the experimentally measured values for l expressed in the time units necessary for one arithmetic operation: they stretch from 500 on a Cray T3D to $3 \cdot 10^5$ on a Parsytec GCel. Therefore, rule (18) should be expected to improve performance for a broad range of block lengths on diverse parallel machines.

Composition on Linear Pipeline. Yet completely different assessment applies for the MPI implementation on a Parsytec GCel (MPICH Version 1.1.0), where scan is realized by a *linear shift* across the processors, with buffered send-receive exchanges. Such scan requires a time of:

$$(p - 1) \cdot (t_s + m \cdot t_w) + p \cdot m$$

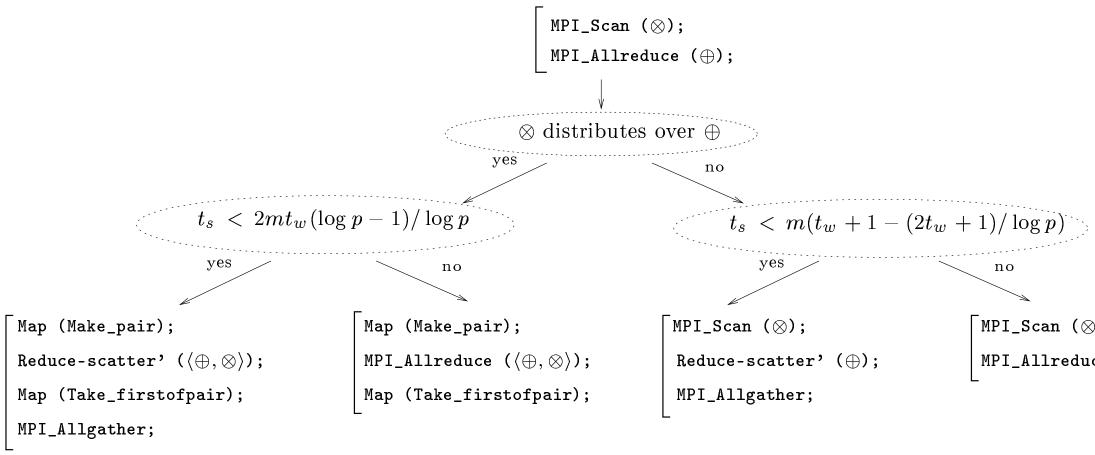

Fig. 5. Design alternatives for transforming a scan followed by a reduction

which is suboptimal, and works well only for small processor configurations. Since, in this MPI version, reduction is realized using the hypercube-like algorithm, it leads to an assessment of the composition rules which is completely different from what we had before.

Indeed, the scan-reduce composition rule improves the performance if: $p-1 \;>\; 2 \cdot \log p$, which is true for all sufficiently big configurations. Note that the improvement factor increases substantially with the number of processors (this will be demonstrated also experimentally in Subsection VII-F, see Figure 6).

In contrast, two consecutive scans can be executed in a pipelined manner, with a time of:

$$p \cdot (t_s + m \cdot t_w) + (p+1) \cdot m$$

whereas the scan obtained by applying composition rule (18) requires a time:

$$(p-1) \cdot (t_s + 2 \cdot m \cdot t_w) + 2 \cdot p \cdot m$$

Thus, the rule application pays off only if:

$$t_s \;>\; (p-2) \cdot m \cdot t_w + (p-1) \cdot m$$

which is almost never true. In fact, composition rule (18) leads to a slow-down of up to 2, which we indeed observed in our experiments! Thus, the effect of the scan-scan composition rule for the linear pipeline is quite the opposite of what it brings for the hypercube-like implementation of scan.

D. Estimates for Decomposition and Combined Rules

Decomposition. Let us consider the right-hand side of decomposition rule (37) on a hypercube. Since the length of communicated segments in stage *allred-parts* decreases from $m/2$ to m/p in $\log p$ steps, this stage requires time:

$$\sum_{i=1}^{\log p} (t_s + m \cdot (t_w + 1)/2^i)$$

After the first stage, each processor keeps a segment of length m/p; these segments are gathered at the second stage to the result of length m in each processor, in time

$$\sum_{i=1}^{\log p} (t_s + m \cdot t_w/2^i)$$

The total time is:

$$2 \cdot t_s \cdot \log p + m \cdot (2 \cdot t_w + 1)(p-1)/p$$

Compared to the time of reduction (43) on the left-hand side of the rule, the performance is improved if for sufficiently large configurations, i.e., big values of p:

$$t_s \;<\; m \cdot (t_w + 1 - (2 \cdot t_w + 1)/\log p)$$

Thus, the reduction decomposition should pay off for big blocks in the processors. This result is also intuitively clear: the bigger the blocks, the more redundant computations and computations are saved due to decomposition at the price of extra start-up costs.

Combined Rule. For composition of the form: *allred-seg* $(\oplus) \circ$ *pref-seg* $(\otimes)$, application of combined rule (41) leads to a new time of:

$$2 \cdot t_s \cdot \log p + 3 \cdot m \cdot (t_w + 1) \cdot (p-1)/p$$

which, when compared to the initial time:

$$\log p \cdot (2 \cdot t_s + m \cdot (2 \cdot t_w + 3))$$

brings an improvement in practically all cases.

E. Choice of Design Alternatives

Figure 5 summarizes our considerations for a composition of scan and reduction. When spotting pattern *allred-seg* $(\oplus) \circ$ *pref-seg* $(\otimes)$ in a BMF algorithm, or its equivalent in an MPI program,

```
MPI_Scan ( ⊗ );
MPI_Allred ( ⊕ );
```

we can either leave it unchanged, or transform it using the design rules introduced in this paper, in one of the three following ways:

- composition rule (39) results in the time complexity:

$$\log p \cdot (t_s + m \cdot (2 \cdot t_w + 3))$$

- combined rule (41) yields time:

$$2 \cdot t_s \cdot \log p + 3 \cdot m \cdot (t_w + 1) \cdot (p-1)/p$$

which is better than the first solution if $t_s < 2 \cdot m \cdot t_w \cdot (\log p - 1)/\log p$, for sufficiently big p.

- One more possibility is to decompose the reduction stage by (37). This design decision yields a slower solution than after applying either the composition or the combined rule. However, the decomposition rule does not impose an additional restriction of distributivity upon the involved operators and, thus, it can be applied in a broader case. As shown before, decomposition improves the performance if:

$$t_s < m \cdot (t_w + 1 - (2 \cdot t_w + 1)/\log p)$$

Figure 5 captures the discussed design alternatives for composing a scan with a reduction in the form of a decision tree. The best design decision is obtained by checking the design conditions which depend either on the problem properties, e.g., the distributivity of operators, or on the characteristics of the target machine (number of processors, speed of the channels, etc.).

For example, in the MSS case study, the distributivity condition holds, which brings us into the left subtree in Figure 5. Since the block size for MSS is very small, the second condition yields "no", and thus we end up with the second (from left to right) design alternative.

Note that the form of conditions in the tree of alternatives may change for a different implementation of the involved collective operations on the same machine.

F. Machine Experiments

Composition in MSS. As a consequence of analytical estimates, we should expect that, of our two target programs for the MSS problem, both of logarithmic time complexity on a hypercube and of linear complexity on a pipeline, program (31) should do better than program (28).

We have implemented both programs in MPI on a Parsytec GCel-64, whose topology is a 2D torus, but the B004 router makes the communication speed largely insensitive to the distance between processors. The three plots in Figure 6 demonstrate the correspondence between the run time before and after optimization, depending on the number of processors, for three different block sizes. Since the absolute times of both scan and reduction in case of *mss* are very small, we also made measurements by storing blocks of data in the processors. In other words, our programs solve several *mss* problems simultaneously; this way we diminish possible accidental effects in the measurements.

The first observation is that the general trend is stable across different block sizes, which provides confidence in our analytical estimates. For small configurations, the overhead of extra computations is larger than the savings due to the reduced communication costs. The time improvement becomes quickly about 2 on 32 processors and increases further on the larger configurations. Whereas this fact confirms both the power of the composition rules and the correctness of the performance estimates, the increasing gap between two curves is a reflection of the situation explained in the previous subsection: extremely inefficient implementation of scan comparing to reduction in the MPI version used in our experiments.

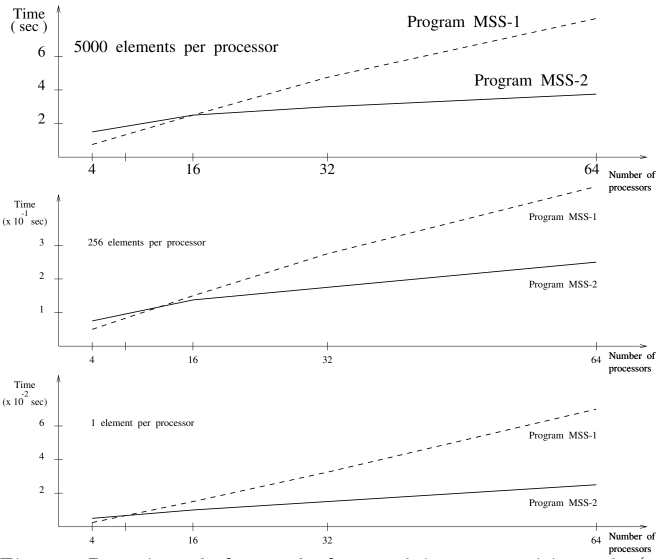

Fig. 6. Run times before and after applying composition rule (12) for the MSS problem.

Decomposition of Reduction. Our experimental results with decomposition rule (37) on a 64-processor Parsytec machine[1] are presented in Figure 7, and they confirm the analytical estimates of Subsection VII-D. Both axes of the plot are logarithmic, and thus the run time improvement achieved by decomposition is substantial.

We have compared the run times of the standard `MPI_Allred` and two decomposed implementations. Since the standard `MPI_Reduce_scatter` is implemented inefficiently on the machines available to us (Parsytec and Cray T3E), we have implemented our `Red-scatter'` as a sequence of swaps given by the first stage of (38). For the gathering step, we have experimented with both the standard `MPI_Allgather`, and the self-written `DH-Allgather` implementation which proceeds exactly as given by the second stage of (38).

We observe that the decomposed reduction improves upon the standard implementation beginning from the segment lengths of about 100 for the self-written gathering, and beginning from the lengths of about 6000 if the standard `MPI_Allgather` is used.

[1] Rolf Rabenseifner at the University of Stuttgart has implemented an analogous decomposed reduction algorithm on an arbitrary number of processors. His experiments on a Cray T3E, which is a 3D torus of processors, exhibit a behaviour very similar to our Figure 7.

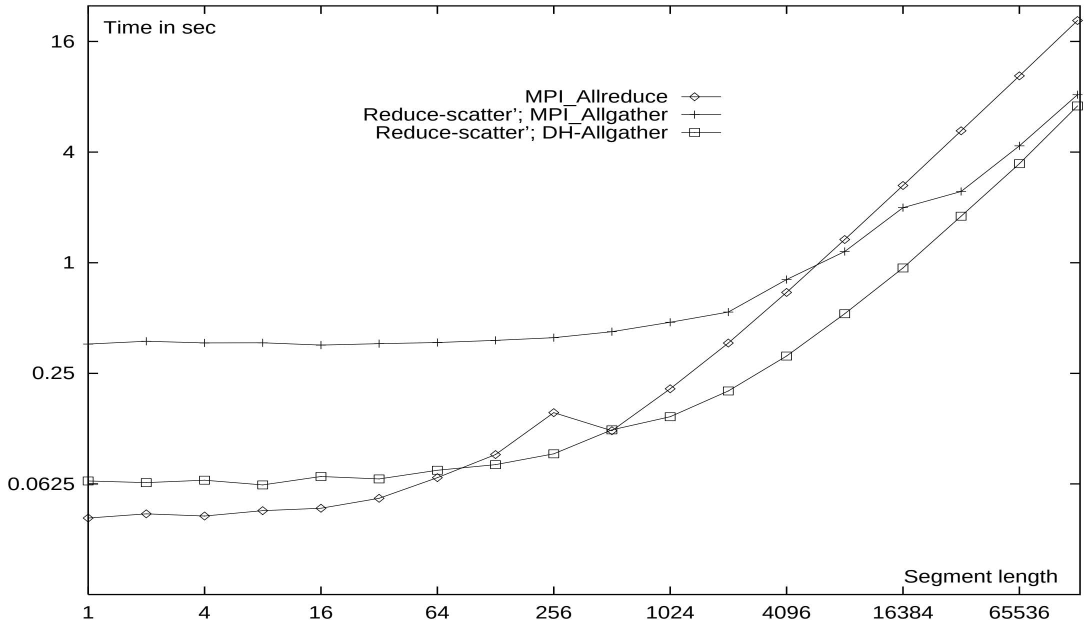

Fig. 7. Reduction decomposition on 64 processors

VIII. Comparison to Related Work

Previous research on message passing with collective operations has concentrated either on expressing *applications* in terms of collective operations (see, e.g., the PLAPACK library for linear algebra [22] or algorithms for computational geometry [11], both based on exclusively collective communications), or on efficient *implementations* of particular collective operations on various parallel architectures (see, e.g., [20]).

In this paper, we build on previous 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.

To the best of our knowledge, our work is the first attempt to put an architecture-independent SPMD language, consisting of sequential computations and MPI collective operations, on a solid formal and methodical base. We have sketched an optimization method, based on transformation rules, augmented with a cost calculus. 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.

The wide applicability of the approach is confirmed also by our most recent studies where we substantially extend the set of transformation rules (including, e.g., the rules for broadcast) [23] and apply our rules in a skeleton-based programming environment [24].

Transformation work related to ours has been carried out in the formal setting of BMF: a version of Theorem 1 in the sequential case was proved and used in [25], [17]. An analog of our derived operation $red\,(\langle\oplus,\otimes\rangle)$, called *recur-reduce*, has been used to parallelize linear recurrences [26] and more recently in our own group to tabulate parallel implementations of linearly recursive functional programs [27]. Our rule (12) differs in that it aims directly at the scan-reduce composition, and is arguably more convenient for the use in both algorithm design and practical MPI programming.

To the best of our knowledge, the composition rule for two scans in Theorem 2 is new; the target expression is similar to *recur-prefix* [26].

Our case study gives a new derivation of a parallel solution for the maximum segment sum problem. We have established a connection between two parallel solutions: an application of Theorem 1 for suffix leads to a solution similar to [26] which, by a further application of Theorem 1 for prefix, is transformed into the solution from [13], [28], [16]. This latter solution is usually more efficient since it consists of just one collective operation, but it has been obtained before by relying on intuition. Our derivation combines the advantages of both solutions, and it is carried out systematically. Moreover, by considering an arbitrary bounded number of processors, we get a cost-optimal solution [19].

The decomposed implementation of reduction has been discussed elsewhere [29], [20], but is still rarely used in practice. Our contributions are: embedding the decomposition rule into a formally-based design methodology, deriving an efficient hypercube implementation, providing an exhaustive performance study, and experimenting with it in the MPI setting. Our experiments show that the decomposition schema may be advantageous not only for the global reduction as has been suggested in the previous work but also for the reduction with the result in one processor, i.e. `MPI_Reduce`.

IX. Conclusions

In this paper, we take a step towards making parallel programming a more systematic, formally-based activity. Our contribution is a collection of design rules for the practically used message passing programming standard, MPI, which are developed on a solid basis of the Bird-Meertens formalism.

We address the trade-off between performance, provable correctness and portability, by distinguishing two phases in the program design process: 1) algorithm design proceeds in an architecture-independent way, using correctness-preserving transformation rules; 2) program design takes into account the details of the MPI implementation on a particular architecture/machine, to determine the impact of particular design decisions on the target performance. Thus, we draw a border line between the architecture-independent algorithmic considerations and the performance-guided program tuning.

Our work demonstrates that compositionality in parallel software engineering is not straight-forward: it cannot be guaranteed that the sequential composition of the best parallel implementations of two program stages yields again optimal parallelism. Instead, one must face the possibility that an individual implementation effort for compositions is called for.

Therefore, a good MPI implementation should not only provide efficient collective operations, but also try and apply optimizing rules for (de)composition, which would be suitable for an underlying platform. This requirement actually does not go much farther than, say, loop fusion implemented in modern sequential compilers. We are collaborating with the *Albatross* group at Amsterdam on incorporating our rules into their MPI implementation for WAN-clustered parallel platforms [30]. If portability between machine models is not provided by the compiler, then the application programmer should at least be aware of the model he/she is programming for; switching models may mean reprogramming.

Let us comment on the restrictions of our approach and on possible future work:

- The exclusive use of collective operations arguably restricts the class of target programs covered by the approach. However, the recent work in parallel algorithmics [11], [31], as well as the latest experience in using MPI [32], [33], suggest that a wide variety of practical applications can be programmed efficiently with collective operations. The future work should identify the situations where the current set of collective operations does not suffice and suggest possible extensions.
- In the algorithm design phase, the critical question is the order in which transformations should be applied. There is still room for intuition and eureka steps here, but the design rules like those we presented can help a lot: e.g., the presence of two collective operations in MSS program (28) has been used as a hint that the corresponding composition rule should be applied next.
- Embedding design rules into an MPI implementation requires checking the rules' applicability conditions like associativity and distributivity. Since that could be a hard or even infeasible task to do automatically, it should probably remain a responsibility of the programmer. Note that the user has already similar obligations nowadays when writing MPI programs with self-defined base operators, whose associativity and commutativity is required by the MPI standard.

Our further work aims at extending the set of semantically sound and practically useful transformation rules for collective operations, which can be employed either by the user in the design process, or by future MPI implementations.

X. Acknowledgements

I am grateful to C. Lengauer for many helpful comments on this manuscript. H. Bischof helped a lot in the experiments on the Parsytec system. Special thanks are due to R. Rabenseifner for an extensive discussion on the decomposed reduction and for sharing his experimental data on a Cray T3E. Thanks to J.-F. Collard, C. Herrmann, B. Lisper, S. Pelagatti and C. Wedler for discussing earlier versions of this manuscript, and to J. Hill for checking the BSP cost estimation.

The anonymous referees were very helpful in improving the quality of presentation.

This work is part of the DFG-funded project *RecuR2*, and has been partially supported by the DAAD exchange programmes *Kista* and *Vigoni*.

References

[1] D. Skillicorn, *Foundations of Parallel Programming*, Cambridge University Press, 1994.

[2] S. Gorlatch, "Abstraction and performance in the design of parallel programs," Habilitation Thesis. Universität Passau. MIP-9802, 1998.

[3] Bruno Bacci, Marco Danelutto, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi., "P^{3}L: A structured high level programming language and its structured support," *Concurrency: Practice and Experience*, vol. 7, no. 3, pp. 225–255, 1995.

[4] M. I. Cole, *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*, Pitman, 1989.

[5] J. O'Donnell and G. Rnger, "A methodology for deriving parallel programs with a family of abstract parallel machines," in *Parallel Processing. Euro-Par'97*, C. Lengauer, M. Griebl, and S. Gorlatch, Eds., Lecture Notes in Computer Science 1300, pp. 661–668. Springer-Verlag, 1997.

[6] B. Jay, M. Cole, M. Sekanina, and P. Steckler, "A monadic calculus for parallel costing of a functional language of arrays," in *Parallel Processing. Euro-Par'97*, C. Lengauer, M. Griebl, and S. Gorlatch, Eds., Lecture Notes in Computer Science 1300, pp. 650–661. Springer-Verlag, 1997.

[7] W. Gropp, E. Lusk, and A. Skijellum, *Using MPI: Portable Parallel Programming with the Message Passing*, MIT Press, 1994.

[8] William F. McColl, "Scalable computing," in *Computer Science Today*, J. van Leeuwen, Ed., Lecture Notes in Computer Science 1000, pp. 46–61. Springer-Verlag, 1995.

[9] D. Culler et al., "LogP: Towards a realistic model of parallel computation," in *Proc. 4th ACM Conf. on Principles and Practice of Parallel Prog. (PPoPP'93)*. 1993, pp. 1–12, ACM Press.

[10] D. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Computing Surveys*, 1998.

[11] X. Deng and N. Gu, "Good programming style on multiprocessors," in *Proc. IEEE Symp. on Parallel and Distributed Pro-*

cessing (SPDP'94). 1994, pp. 538–543, IEEE Computer Society Press.

[12] R. Bird, "Lectures on constructive functional programming," in *Constructive Methods in Computing Science*, M. Broy, Ed., NATO ASI Series F: Computer and Systems Sciences. Vol. 55, pp. 151–216. Springer Verlag, 1988.

[13] M. Cole, "Parallel programming with list homomorphisms," *Par. Proc. Letters*, vol. 5, no. 2, pp. 191–204, 1994.

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

[15] J. Bentley, "Programming pearls," *Comm. ACM*, vol. 27, pp. 865–871, 1984.

[16] D. Smith, "Applications of a strategy for designing divide-and-conquer algorithms," *Science of Computer Programming*, vol. 8, no. 3, pp. 213–229, 1987.

[17] D. Swierstra and O. de Moor, "Virtual data structures," in *Formal Program Development*, B. Möller, H. Partsch, and S. Schuman, Eds., Lecture Notes in Computer Science 755, pp. 355–371. Springer-Verlag, 1993.

[18] V. Kumar et al., *Introduction to Parallel Computing*, Benjamin/Cummings Publ., 1994.

[19] Sergei Gorlatch, "Systematic efficient parallelization of scan and other list homomorphisms," in *Euro-Par'96: Parallel Processing, Vol. II*, Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, Eds., Lecture Notes in Computer Science 1124, pp. 401–408. Springer-Verlag, 1996.

[20] Robert van de Geijn, "On global combine operations," *J. Parallel and Distributed Computing*, vol. 22, pp. 324–328, 1994.

[21] J. Hill and D. Skillicorn, "The BSP Tutorial at Euro-Par'97," University of Passau, 1997.

[22] Robert van de Geijn, *Using PLAPACK: Parallel Linear Algebra package*, Scientific and Engineering Computation Series. MIT Press, 1997.

[23] Sergei Gorlatch, Christoph Wedler, and Christian Lengauer, "Optimization rules for programming with collective operations," in *13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing (IPPS/SPDP'99)*, Mikhail Atallah, Ed. 1999, pp. 492–499, IEEE Computer Society Press.

[24] Bruno Bacci, Sergei Gorlatch, Christian Lengauer, and Susanna Pelagatti, "Skeletons and transformations in an integrated parallel programming environment," in *Parallel Computing Technologies (PaCT-99)*. 1999, LNCS 1662, pp. 13–27, Springer-Verlag.

[25] R. Bird, "Algebraic identities for program calculation," *The Computer J.*, vol. 32, no. 2, pp. 122–126, 1989.

[26] W. Cai and D. Skillicorn, "Calculating recurrences using the Bird-Meertens formalism," *Parallel Processing Letters*, vol. 5, no. 2, pp. 179–190, 1995.

[27] C. Wedler and C. Lengauer, "Parallel implementations of combinations of broadcast, reduction and scan," in *Proc. 2nd Int. Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'97)*, G. Agha and S. Russo, Eds. 1997, pp. 108–119, IEEE Comp. Soc. Press.

[28] Z. Hu, H. Iwasaki, and M. Takeichi, "Formal derivation of efficient parallel programs by construction of list homomorphisms," *ACM TOPLAS*, vol. 19, no. 3, pp. 444–461, 1997.

[29] G. Fox et al., *Solving Problems on Concurrent Processors*, vol. 1, Prentice-Hall, 1988.

[30] Thilo Kielmann, Henri E. Bal, and Sergei Gorlatch, "Bandwidth-efficient collective communication for clustered wide area systems," Submitted to IPDPS'2000, available at: http://www.cs.vu.nl/ kielmann/papers/ipdps.ps.gz.

[31] Aske Plaat, Henri E. Bal, and Rutger F.H. Hofman:, "Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects," in *Proceedings of HPCA-5*, 1999, To appear.

[32] Thilo Kielmann, Rutger F.H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A.F. Bhoedjang., "MagPIe: MPI's collective communication operations for clustered wide area systems.," in *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, 1999, To appear.

[33] P.S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann Publ., 1997.

Sergei Gorlatch received his PhD in 1984 from Glushkov Institute of Cybernetics, Kiev, Ukraine. He was actively involved into the architecture and software design for ES-1766 – the first distributed memory MIMD-computer manufactured serially in the former Soviet Union. In 1991-92 he worked as a Humboldt Research Fellow at the Technical University of Munich, Germany. Since 1992 Dr. Gorlatch has been with the University of Passau, Germany. In 1998 he received his Habilitation (post-doctoral degree). His current research interests include parallel algorithms, programming methodology and formal methods for parallel and distributed systems, and performance evaluation.