Parallel Processing Letters,
© World Scientific Publishing Company

A GENERIC MPI IMPLEMENTATION FOR A DATA-PARALLEL SKELETON: FORMAL DERIVATION AND APPLICATION TO FFT *

SERGEI GORLATCH and HOLGER BISCHOF
University of Passau, D–94030 Passau, Germany
Email: gorlatch@fmi.uni-passau.de

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

We derive a provably correct, architecture-independent family of parallel implementations for a class of data-parallel algorithms, called DH (distributable homomorphisms). The implementations are well-structured SPMD programs with group-wise personalized all-to-all exchange, directly realizable in MPI. As a case study, we systematically adjust the mathematical specification of the Fast Fourier Transform (FFT) to the DH format and, thereby, obtain a generic SPMD implementation for FFT. The target program includes FFT solutions used in practice – the binary-exchange and the 2D- and 3D-transpose – as special cases.

Keywords: Formal Methods; Skeletons; Bird-Meertens Formalism (BMF); Fast Fourier Transform (FFT); Program Derivation; Message Passing Interface (MPI).

1. Introduction

A way to improve the current practice of low-level, error-prone and hardly portable parallel programming is offered by identifying adequate algorithmic schemas (*skeletons* [6]), and deriving their provably correct, efficient implementations in a suitable formalism. The task of the user is then to express a particular application as an instance of a suitable skeleton, which immediately leads to a concrete target program. Related work includes skeleton programming systems [1,4], and optimization of skeleton compositions [9]. This follows the trend towards well-structured parallel solutions with predictable performance, which is exemplified by the BSP-like models [14], coarse-grain algorithms [7], and collective primitives of MPI [12].

This paper applies the skeleton approach to the design of data-parallel programs [5]. Our contribution is a formal derivation of a generic parallel implementation for a skeleton called DH (Distributable Homomorphism) [8]. The target program has a well-structured SPMD format, with stages of two kinds: (1) parallel computations in the processors, and (2) collective, group-wise communications; both kinds are directly implementable under MPI. Once derived, the generic implementation can be used for a particular application by customizing the skeleton in a suitable way. As

*A preliminary version [10] was presented in the FMMPTA Workshop at IPPS'97 (Geneva).

an illustration, a mathematical specification of FFT is expressed in the DH format, thus providing a generic parallel program for FFT. This systematically developed implementation covers the practically used solutions for FFT, which have been traditionally based on *ad hoc* design decisions. We report the experimental results of the derived FFT implementation on a 64-transputer network.

2. Data-Parallelism and DH

Data-parallel programs compute functions on data aggregates like arrays, with parallelism achieved by distributing the data among the processors, and organizing their simultaneous work on the data parts. For a given function h, defined on a monolithic data structure, its typical data-parallel implementation reads as follows:

$$h \; = \; out \;\circ\; h_{in}^{out} \;\circ\; in \tag{1}$$

with three phases connected by means of functional composition $\circ$: 1) function *in* partitions the input data, 2) central phase, h_{in}^{out}, called the *distributed phase*, takes data arranged by *in* and yields result again in a partitioned (possibly differently) form, 3) function *out* flattens the partitioned data into a monolithic form.

Function h_{in}^{out} in (1) comprises the "useful" computations and communications of a data-parallel algorithm. Several such algorithms are often combined within a single application, supplying and consuming partitioned data from each other, possibly with a repartitioning. Therefore, one is usually interested in finding a good distributed implementation h_{in}^{out}, with *in* and *out* viewed as the partitioning information. We will address the implementation problem for a class of data-parallel algorithms which are described by a skeleton introduced below.

Our formal base is a subset of the functional *Bird-Meertens formalism* (BMF) [2], defined on *powerlists* [15] – non-empty lists whose length is a power of 2, with list concatenation $+\!\!+$. Two functionals with obvious data-parallel semantics are defined:

$map\ f$	*map* of an unary function f : $map\ f\,[x_1, \ldots, x_n] \stackrel{\text{def}}{=} [\,f\,x_1\,, \ldots,\, f\,x_n\,]$;
$zip\,(\odot)$	combines elements of two lists of equal length with operation $\odot$, $zip\,(\odot)\ (\,[x_1, \ldots, x_n]\,,\,[y_1, \ldots, y_n]\,) \stackrel{\text{def}}{=} [\,(x_1 \odot y_1), \ldots, (x_n \odot y_n)\,]$.

We study a class of functions which can be specified as follows [8]:

Definition 1 *For binary operations* $\oplus$ *and* $\otimes$*, the* Distributable Homomorphism (DH) *is function* $h \,=\, (\oplus \updownarrow \otimes)$ *on lists, such that for arbitrary powerlists* x *and* y *of equal lengths:*

$$\begin{aligned} h\,[a] \;&=\; [a] \\ h\,(x +\!\!+ y) \;&=\; zip\,(\oplus)\,(h\,x, h\,y) \;+\!\!+\; zip\,(\otimes)\,(h\,x, h\,y) \end{aligned} \tag{2}$$

Figure 1 illustrates how a distributable homomorphism $h \,=\, (\oplus \updownarrow \otimes)$ is computed on a concatenation of two lists; dashed arrows denote replication of the partial results.

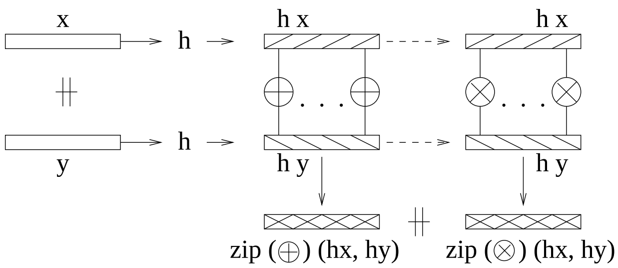

Figure 1: Distributable homomorphism h: computing $h(x +\!\!+\, y)$

Since operations $\oplus$ and $\otimes$ in Definition 1 are parameters, DH can be viewed as a skeleton which describes a class of algorithms. As shown in [8], DH is a subclass of list homomorphisms [2]; DH functions include the familiar reduction and scan functions, and are particularly suitable for hypercube implementation. In [11], a dual version of DH – interleaving DH – is introduced which uses operator $\bowtie$ [15] on the right-hand side of (2) instead of concatenation

3. Deriving Distributed Implementation

For an arbitrary DH function h, we are looking for its data-parallel implementation of form (1), suitable for architecture-independent implementation under MPI. The implementation is derived by exclusively using the equalities of BMF, which guarantees the correctness of the target program.

3.1. Nested Lists

To study the parallel implementation of DH, we use nested lists, restricting ourselves to "square" structures. Function $nest^d$ yields, for a list, its (d+1)-dimensional representation, with equal length in all dimensions. For example, in the two-dimensional case, $d = 1$, function $nest^1$ creates an "$m * m$-matrix", i.e. a list of m sublists, where the length of each sublist is $m = \sqrt{|x|}$. It can be viewed as a restricted version of the popular BLOCK-distribution. We denote by $flat^d$ the left inverse of $nest^d$: it flattens a $(d+1)$-dimensional list.

The type of d-nested lists with m sublists in all dimensions is denoted by $[\alpha]^d_m = [\ldots[\alpha]_m \ldots]_m$. For a list of type $[\alpha]^2_m$, function *map* allows to apply a list function $h : [\alpha] \to [\alpha]$ to each inner list or, in the matrix form, to each row of the matrix:

$$map\ h \begin{bmatrix} [a_{11} & \ldots & a_{1m}], \\ [a_{21} & \ldots & a_{2m}], \\ \vdots & & \vdots \\ [a_{m1} & \ldots & a_{mm}] \end{bmatrix} = \begin{bmatrix} h[a_{11} & \ldots & a_{1m}], \\ h[a_{21} & \ldots & a_{2m}], \\ \vdots & & \vdots \\ h[a_{m1} & \ldots & a_{mm}] \end{bmatrix} \quad (3)$$

We would like to map list functions also along the columns and, in general, along an arbitrary dimension of a multidimensional list. We introduce function *chdim* which, in a d-dimensional list, interchanges two given dimensions, a and b:

$$\begin{aligned}(\; chdim^{(a,b)} \; x \;)_{(i_1,\ldots,i_d)} \quad &\stackrel{\text{def}}{=} \quad x_{(j_1,\ldots,j_d)}, \\ \text{where} \quad & 1 \leq a,b \leq d; j_a = i_b,\, j_b = i_a \text{ and } j_k = i_k \text{ for all other } k.\end{aligned}$$

Note that the usual 2D matrix transposition is a special case of the *chdim* function: $tr \; = \; chdim^{(1,2)}$. We use *chdim* to define a generalized version of *map*, functional *mapdim*, for lists of type $[\alpha]^d$, function $h : [\alpha] \to [\alpha]$ and $1 \leq i \leq d$, which allows one to map a list function along an arbitrary dimension i of a d-dimensional list:

$$mapdim^{(i,d)} \; h \;\; \stackrel{\text{def}}{=} \;\; chdim^{(i,d)} \;\circ\; map^{(d-1)} \; h \;\circ\; chdim^{(i,d)} \tag{4}$$

Note that $mapdim^{(d,d)} \; = \; map^{(d-1)}$, thus (3) illustrates $mapdim^{(2,2)} \; = \; map$. Mapping along the columns of a two-dimensional list is captured by $mapdim^{(1,2)}$:

$$mapdim^{(1,2)} \;\; h \;\; \begin{bmatrix} [a_{11} & \ldots & a_{1m}], \\ [a_{21} & \ldots & a_{2m}], \\ \vdots & & \vdots \\ [a_{m1} & \ldots & a_{mm}] \end{bmatrix} \;\; = \;\; tr \left(\begin{bmatrix} h[a_{11} & \ldots & a_{m1}], \\ h[a_{12} & \ldots & a_{m2}], \\ \vdots & & \vdots \\ h[a_{1m} & \ldots & a_{mm}] \end{bmatrix} \right) \tag{5}$$

3.2. Distributed Implementation

Let us introduce a notation for multiple functional composition, $\overset{l-1}{\underset{i=0}{\bigcirc}}$, which has the same meaning for $\circ$ as $\sum_{i=0}^{l-1}$ has for $+$. Note that $\circ$ is backward, so that, e.g., $\overset{1}{\underset{i=0}{\bigcirc}} h_i \; = \; h_0 \;\circ\; h_1$ will execute starting from h_1.

The following theorem provides a generic data-parallel implementation of format (1) for an arbitrary DH function.

Theorem 1 (Implementation of DH) *Every distributable homomorphism h on a list of length 2^l can be computed for each $d = 0, \ldots, l-1$ as follows:*

$$h \;\; = \;\; flat^d \circ \;\; (\overset{d+1}{\underset{i=1}{\bigcirc}} mapdim^{(i,d+1)} \; h \;) \;\; \circ \; nest^{\,d} \tag{6}$$

Proof. *Induction with BMF-transformations; see Appendix A.* $\square$.

The right-hand side of (6) describes a family of parallel implementations for a DH, one implementation for each $d = 1, \ldots, l-1$:

$d = 0$: corresponds to the sequential execution.

$d = 1$: prescribes the 2D data arrangement, i.e. a matrix, on which two *mapdims* are executed, with two transpositions $chdim^{(1,2)}$; see Figure 2. Note that for $i = 2$, both *chdims* disappear since $chdim^{(2,2)}$ is the identity function.

...

$d = l-1$: input is arranged as a logical l-dimensional hypercube; computation of h proceeds in l steps: at step d, function h is computed independently on $2^{(l-1)}$ pairs of elements which are neighbours in the dimension d of the hypercube.

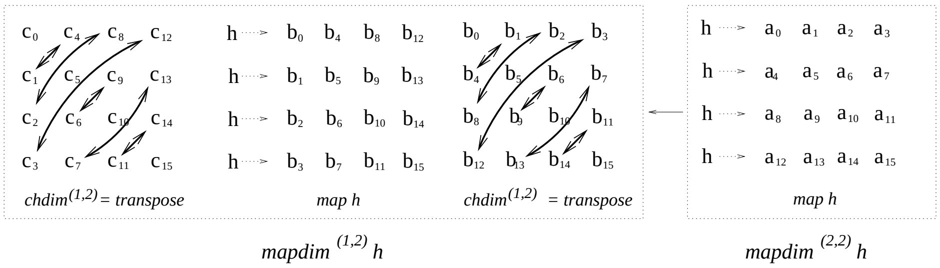

Figure 2: Implementation (6) in the two-dimensional case, $d = 1$.

Thus, Theorem 1 describes a spectrum of algorithms which are usually presented independently of each other in the literature. For instance, case $d=1$ is known as the 2D-transpose schema, and case $d = l-1$ as the binary-exchange schema.

3.3. Optimization

Let us consider the central stage of (6). According to definition (4), there are two *chdim*s between every pair of *map*s in the multiple composition. We can push a half of these data movements towards the end of the expression as follows:

Theorem 2 (Communications Rearrangement)

$$\bigcirc_{i=1}^{d+1} mapdim^{(i,d+1)}\, h \;=\; \bigcirc_{i=1}^{d} chdim^{(i,i+1)} \;\circ\; \bigcirc_{i=1}^{d+1} \left(map^{d}\, h \,\circ\, chdim^{(i,d+1)} \right) \tag{7}$$

Proof. *See Appendix B.* $\square$.

The right-hand side of (7) is a composition of two parts, the second of which, $shiftright \stackrel{\text{def}}{=} \bigcirc_{i=1}^{d} chdim^{(i,i+1)}$, does solely a cyclic shift of the dimensions to the right; it guarantees that the output list is distributed the same way as the input. Following the custom in practice, we can incorporate this rearrangement into the annotation of the output distribution.

Therefore, for each $0 \le d \le l-1$ there is a data-parallel implementation of a distributable homomorphism h:

$$\begin{aligned} h \;&=\; out \circ h_{in}^{out} \circ in, \\ \text{where}\quad in \;&=\; nest^{d}, \\ h_{in}^{out} \;&=\; \bigcirc_{i=1}^{d+1} \left(map^{d}\, h \circ chdim^{(i,d+1)} \right), \\ out \;&=\; flat^{d} \circ shiftright \end{aligned} \tag{8}$$

Functions *in* and *out* specify that the distributed phase h_{in}^{out} takes the data as a $(d+1)$-dimensional array and yields the result array with shifted dimensions.

3.4. Bounded Number of Processors

The amount of parallelism in generic program (8) is expressed by map^d, and grows as $n^{d/d+1}$ with problem size n and dimensionality d. In practice, we must map this potentially unbounded parallelism onto an arbitrary but bounded number of processors, p. This is routinely achieved by assigning to each processor a block of data (vectors) rather than a single vector. In our formalism, the program can be transformed by introducing an additional level of list nesting for blocks, and using two equalities: $flat \circ nest = id$, and the map-promotion law of BMF [2]:

$$
\begin{aligned}
map\, h \;=\; map\, h \circ flat \circ nest \;&=\; flat \circ map\,(map\, h) \circ nest && (9)\\
&=\; flat \circ map\_par\,(map\_seq\, h) \circ nest && (10)
\end{aligned}
$$

There are two representations of the right-hand side: as a pure BMF expression in (9), and with annotations in (10) prescribing how *map*s are to be implemented: *map_par* specifies that computations over the blocks are executed in parallel, whereas *map_seq* specifies a sequential implementation within a block. For more details and another application of the BMF blocking technique, see [8].

We adopt a simple block distribution of data. For each d, $1 \leq d \leq \log p$, p processors constitute a virtual d-dimensional structure, with $p^{\frac{1}{d}}$ processors in each dimension†. No assumption is made about the inter-processor connections because we aim at an architecture-independent solution. First d dimensions of data are distributed evenly among processors, the last dimension is left undistributed. Therefore, each processor stores a $(d+1)$-dimensional block of size: $(n^{\frac{1}{d+1}}/p^{\frac{1}{d}}) \times \cdots \times (n^{\frac{1}{d+1}}/p^{\frac{1}{d}}) \times n^{\frac{1}{d+1}}$. Such kind of distribution is often used in practice; it is annotated in HPF as (BLOCK, ... , BLOCK, *) where the number of the BLOCK-instances is d, and $*$ denotes the undistributed dimension. The dimensionality d and, thus, the number of different implementations of (8) is restricted by $\log p$.

The generic implementation of the DH-skeleton h on a bounded number of processors with square block distribution reads as follows:

```
Program Dist-Impl (d, h, mypid);
/- Input:(BLOCK,...,BLOCK, * )
  For i := d+1 To 1 Do
    All-To-All (group (i,d,mypid));                  (11)
    Local-Transpose (i,d);
    Map_seq h;
  End-Do;
/- Output:(BLOCK,...,BLOCK, * ,BLOCK)
```

The structure of the SPMD program `Dist-Impl` corresponds directly to (8) after applying (9). Functions *nest* and *flat* are expressed in (11) as annotations of the input and output data distributions, marked by `\-` . The distributed phase proceeds in $(d+1)$ stages. Collective group-wise communication `All-To-All` together with

†Can be generalized for arbitrary rectangular blocks [3].

`Local-Transpose` implement $chdim^{(i,d+1)}$ of (8). Computation $map\_par\ (map\_seq\ h)$ from (10) is implemented as follows: map_par is realized by the SPMD structure of (11) which is executed in all processors in parallel; map_seq is implemented by `Map_seq` that computes h for each vector of the block in a processor sequentially.

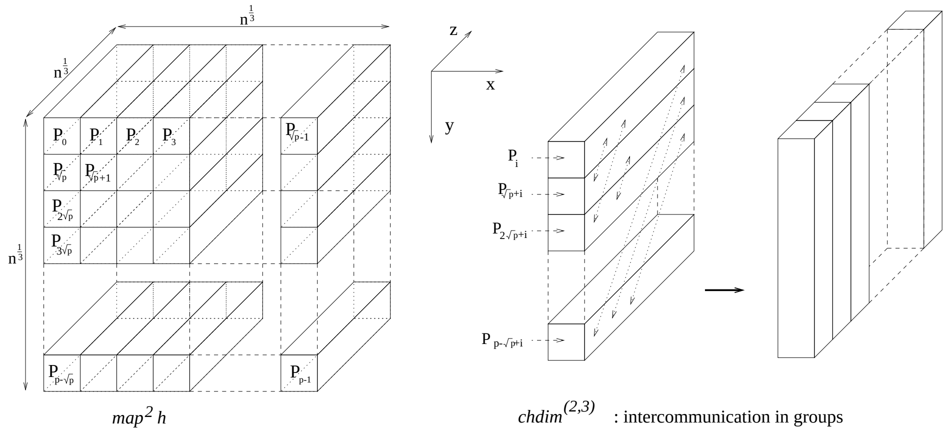

Figure 3: Three-dimensional case, $d = 2$: the transition between the first and the second stage (adapted from [13]).

Case $d = 1$ corresponds as before to a matrix, but now its rows are aggregated in p consecutive blocks, stored in the processors; the second dimension is not distributed. The three-dimensional case, $d = 2$, is shown in Figure 3, adapted from [13]. It illustrates the transition between the first two of altogether three stages in the 3D case on p processors. In the left part of the figure, h is computed in all processors simultaneously along the last dimension, z. The right part of the figure illustrates the beginning of the next stage, associated with dimension y: it starts with interchanging dimensions z and y, which is done by the all-to-all communication within each group of processors located in one y-column.

Case $d = \log p$ defines the hypercube implementation, where all-to-all exchange reduces to pairwise communications along one of the hypercube dimensions.

4. Application to FFT

4.1. Specification and skeleton customization

The Fast Fourier Transform of input list $x = [x_0, \ldots, x_{2m-1}]$ can be specified as follows (see [13]), with $u = [x_0, x_2, \ldots, x_{2m-2}]$ and $v = [x_1, x_3, \ldots, x_{2m-1}]$:

$$\left(\, FFT\,x\,\right)_i = \begin{cases} (FFT\,u)_i \,+\, \omega_{2m}^{i}\,(FFT\,v)_i & ,i < m \\ (FFT\,u)_{i-m} \,-\, \omega_{2m}^{i-m}\,(FFT\,v)_{i-m} & ,i \geq m \end{cases} \tag{12}$$

where $\omega_m = e^{2\pi\sqrt{-1}/m}$, and $i = 0, \ldots, 2m-1$.

To use our generic implementation (11) in case of FFT, we must adjust specification (12) to DH skeleton (2), i.e. find customizing operations $\oplus$ and $\otimes$. This can be done in three systematic steps (see [3,9] for details): 1) by inverting the input list, we change the splitting of x from interleaving as in (12) to concatenation as required by the DH skeleton; 2) we augment function FFT by two auxiliary functions that maintain the values necessary for computing the twiddle factors ω_{2m}^{j}; 3) we optimize this solution by maintaining the necessary information for the twiddle factors locally in the processors; that leads to a more complicated function `FFT_adjusted`, but it saves communications and is arguably more efficient [3].

The MPI-like pseudocode for (11) after FFT-customization reads as follows:

```
Program Dist_impl_MPI (...);
   MPI_Cart_create(MPI_COMM_WORLD, ..., &cart_comm);
   For i=d+1 downto 1 do
      MPI_Comm_split(cart_comm, ..., &comm);
      MPI_Alltoall(local_block, ..., comm);
      Local_transpose(i, d);
      For (each vector v in local_block) do
         FFT_adjusted (v)
      End_for;
      MPI_Comm_free (&comm);
   End_for;
```

Let us briefly comment on how (11) is implemented by `Dist_impl_MPI`. We organize groups of processors for `Alltoall` in two steps: initial creation of the virtual d-dimensional processor structure is done by `MPI_Cart_create`; the groups themselves are created by `MPI_Comm_split` as a communicator with name `comm`, and are eliminated after their use by means of `MPI_Comm_free`. The block computations represented in (11) by `Map_seq h` are implemented as a sequential loop over all the vectors in the block, the loop body computes function `FFT_adjusted`.

4.2. Performance estimation and experiments

Due to the well-structured organization of our generic implementation (11), its performance estimation can be done in a systematic way, stage per stage. For a DH function h, its parallel d-dimensional implementation on p processors requires time:

$$t_{par}(h) = c \cdot t_{seq}(h)/p + t_{cart} + (d+1) \cdot (t_{alltoall} + t_{loc-transp} + t_{split-free}) \qquad (13)$$

where the following parameters are used:

$t_{seq}(h)$	time of sequential computation for input size n;
c	coefficient ≥ 1 showing the adjsutment overhead;
t_{cart}	time to create cartesian topology of p processors;

$t_{alltoall}$	time for global exchange in $p^{\frac{d-1}{d}}$ groups of $p^{\frac{1}{d}}$ processors with blocks of $n/p^{\frac{d+1}{d}}$ elements;
$t_{loc-transp}$	time for n/p data movements in local memory;
$t_{split-free}$	time to partition p processors into $p^{\frac{d-1}{d}}$ groups, and to free them.

Let us demonstrate how generic estimation (13) can be applied in the case of FFT. The extra computations within blocks (i.e. in function `FFT-adjusted`) yield factor $c = 1.5$ when compared with original sequential FFT algorithm. Time t_{cart} turns to be negligible comparing to the other parts of (13). The rest of parameters have been benchmarked experimentally.

Figure 4 shows two plots of speed-up on 64 processors depending on the problem size, for different values of d. The left plot shows the speed-up values predicted via formula (13) as explained above; the right plot shows the experimental results. We observe that the accuracy of our prediction is about 15%. Experiments were carried out with MPICH 1.0 on the transputer machine Parsytec GCel64 under OS Parix.

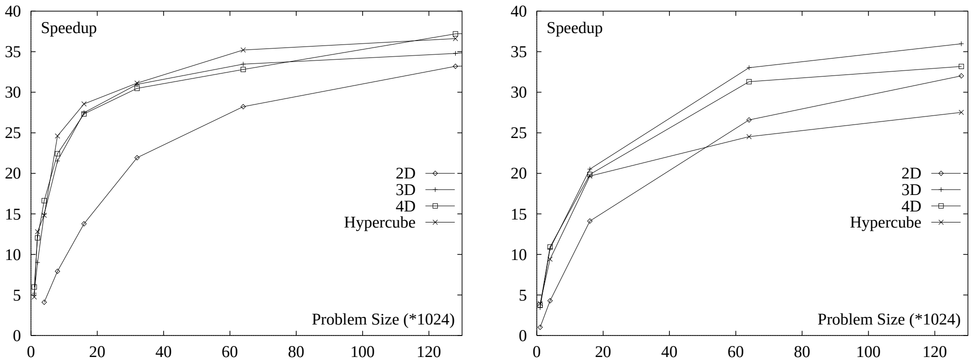

Figure 4: Speedup of d-dimensional FFT on 64 processors

Depending on the particular machine and on the problem size, the optimal value of d can be chosen. The performance analysis of FFT in [13] shows the following trend: as the dimensionality d increases, the communication overhead due to the per-word transfer time in the communication network increases, whereas the overhead due to the start-up delay decreases.

5. Conclusion

The paper demonstrates a systematic, formally based approach to parallel programming. For a class of DH functions, a family of provably correct, parallel implementations is formally derived. We illustrate the use of the approach on the FFT case study; the systematically obtained implementation includes the FFT solutions used in practice as special cases. Interestingly, the decision to invert the input, which we have made in order to meet the DH format, is used *ad hoc* in practice.

A by-product of this work is a better understanding of what kind of algorithms are implementable in "a good programming style" [7], with clearly decoupled computation and collective communication stages. Such programs are easy to understand, and their performance is better predictable. Our solutions are topology-independent and well suited for implementation on MPI, and thereby are portable across many parallel and distributed platforms.

The restriction of our approach is in the specific DH skeleton format, which at the same time allows us to obtain practically relevant results.

Acknowledgments

Thanks to C. Lengauer, C. Herrmann and C. Wedler for technical discussions, and to the anonymous referees for helpful remarks. This work was supported by the EU (INTAS Programme) and DAAD (exchange programmes ARC and PROCOPE).

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

[3] H. Bischof. Formal derivation and implementation of parallel MPI-programs from homomorphisms. M.Sc. Thesis (in German), University of Passau, 1998.

[4] G. Botorog and H. Kuchen. Efficient parallel programming with algorithmic skeletons. In L. Bougé et al., editors, *Parallel Processing. Euro-Par'96*, Lecture Notes in Computer Science 1123, pages 718–731. Springer-Verlag, 1996.

[5] L. Bougé. The data-parallel programming model: a semantic perspective. In A. Darte and G.-R. Perrin, editors, *The Data-Parallel Programming Model*, Lecture Notes in Computer Science 1132, pages 4–26. Springer-Verlag, 1996.

[6] M. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. PhD thesis, University of Edinburgh, 1988.

[7] X. Deng and N. Gu. Good programming style on multiprocessors. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, pages 538–543, 1994.

[8] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bougé et al., editors, *Parallel Processing. Euro-Par'96*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[9] S. Gorlatch. Programming with divide-and-conquer skeletons: An application to FFT. *J. Supercomputing*, 12(1-2):85–97, 1998.

[10] S. Gorlatch and H. Bischof. Formal derivation of divide-and-conquer programs: A case study in the multidimensional FFT's. In D. Mery, editor, *Formal Methods for Parallel Programming: Theory and Applications. Workshop at IPPS'97*, pages 80–94, 1997. Available at http://www.fmi.uni-passau.de/cl/papers/GorBi97.html.

[11] S. Gorlatch and C. Lengauer. (De)Composition rules for parallel scan and reduction. In *Proc. 3rd Int. Working Conf. on Massively Parallel Programming Models (MPPM'97)*. IEEE Computer Press, 1998. Available at http://fmi.uni-passau.de/cl/papers/GorLe97c.html.

[12] W. Gropp, E. Lusk, and A. Skijellum. *Using MPI: Portable Parallel Programming*

with the Message Passing. MIT Press, 1994.
[13] V. Kumar et al. *Introduction to Parallel Computing*. Benjamin/Cummings Publ., 1994.
[14] W. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science 1000, pages 46–61. Springer-Verlag, 1995.
[15] J. Misra. Powerlist: a structure for parallel recursion. *ACM TOPLAS*, 16(6):1737–1767, 1994.

Appendix A : Proof of Theorem 1

Some auxiliary equalities are used as transformation rules [10]:

$$map\ (f \circ g) = map\ f\ \circ\ map\ g \tag{A.1}$$
$$(\oplus \updownarrow \otimes)\ \circ\ flat = red\,(\odot)\ \circ map(\oplus \updownarrow \otimes) \tag{A.2}$$
$$(zip \oplus) \updownarrow (zip \otimes) = nest\ \circ\ red\,(\odot) \tag{A.3}$$
$$(zip \oplus) \updownarrow (zip \otimes)\ z = \left(map\ h\ (z^t)\right)^t \tag{A.4}$$
$$\text{For } a \leq d : map\ (mapdim^{(a,d)}\ f) = mapdim^{(a+1,d+1)}\ f \tag{A.5}$$
For $a \leq (d-1)$, $d \geq 2$:
$$\left(map^{d-1}\ flat\right)\ \circ\ mapdim^{(a,d+1)}\ h = mapdim^{(a,d)}\ h\ \circ\ \left(map^{d-1}\ flat\right) \tag{A.6}$$
$$\text{For } d \geq 2 :\ flat^d = flat^{d-1}\ \circ\ map^{d-1}\ flat \tag{A.7}$$

where $x^t = chdim^{(1,2)}\ x$; $x, y, z \in [\alpha]_{n \times m}$; $h = \oplus \updownarrow \otimes$; $f : [\alpha] \to [\alpha]$
and in (A.2), (A.3): $(\oplus \updownarrow \otimes)\,(x +\!\!\!+\, y) = (\oplus \updownarrow \otimes)\,x\ \odot\ (\oplus \updownarrow \otimes)\,y$.

Let us show that: $h = flat^d\ \circ\ \bigcirc_{i=1}^{d+1} mapdim^{(i,d+1)}\ h\ \circ\ nest^d$.

Proof. Induction on d.
Base case $d = 0$ obvious: $h = flat^0\ \circ\ mapdim^{(1,1)}\ h\ \circ\ nest^0 = h$.
Base case $d = 1$:

$$
\begin{aligned}
& flat\ \circ\ mapdim^{(1,2)}\ h\ \circ\ mapdim^{(2,2)}\ h\ \circ\ nest && \{\ \text{(A.4), Def. of } mapdim\ \} \\
=\ & flat\ \circ\ (zip \oplus) \updownarrow (zip \otimes)\ \circ\ map\ h\ \circ\ nest && \{\ \text{(A.3), (A.2)}\ \} \\
=\ & flat\ \circ\ nest\ \circ\ h\ \circ flat\ \circ\ nest\ =\ h
\end{aligned}
$$

Induction step: let $d > 1$:

$$
\begin{aligned}
& flat^d\ \circ\ \bigcirc_{i=1}^{d+1} mapdim^{(i,d+1)}\ h\ \circ\ nest^d \\
=\ & \quad \{\ \text{Let } A\ =\ flat^{d-1}\ \circ\ \bigcirc_{i=1}^{d-1} mapdim^{(i,d)}\ h, \text{ after (A.7) and (A.6)}\ \} \\
& A\ \circ\ map^{d-1}\ flat\ \circ\ mapdim^{(d,d+1)}\ h\ \circ\ mapdim^{(d+1,d+1)}\ h\ \circ\ nest^d \\
=\ & \quad \{\ \text{(A.1), (A.5)}\ \} \\
& A\ \circ\ map^{d-1}\ flat\ \circ\ map^{d-1} \left(mapdim^{(1,2)}\ h\ \circ\ mapdim^{(2,2)}\ h\right)\ \circ\ nest^d \\
=\ & \quad \{\ \text{Definition of } nest \text{ and (A.1)}\ \} \\
& A\ \circ\ map^{d-1}\ \left(flat\ \circ\ mapdim^{(1,2)}\ h\ \circ\ mapdim^{(2,2)}\ h\ \circ\ nest\right) \circ nest^{d-1} \\
=\ & \quad \{\ \text{Base case } d = 1\ \} \\
& A\ \circ\ map^{d-1}\ h\ \circ\ nest^{d-1} \\
=\ & \quad \{\ \text{Definition of } mapdim\ \}
\end{aligned}
$$

$$flat^{d-1} \circ \bigcirc_{i=1}^{d} mapdim^{(i,d)}\ h \circ nest^{d-1}$$
$$= \quad \{ \text{ Induction hypothesis } \}$$
$$h$$
$\square$.

Appendix B : Proof of Theorem 2

For an arbitrary d, we prove a more general proposition, namely, for all $1 \leq k \leq d$:

$$\bigcirc_{i=k}^{d+1}\left(chdim^{(i,d+1)} \circ map^{d}\ h \circ chdim^{(i,d+1)}\right) = \bigcirc_{i=k}^{d} chdim^{(i,i+1)} \circ \bigcirc_{i=k}^{d+1}\left(map^{d}\ h \circ chdim^{(i,d+1)}\right)$$

Proof. Induction on k which starts with d as the base case.
Base case: $k = d$, obvious. Induction step $k \to k-1$, $k \geq 2$:

$$\bigcirc_{i=k-1}^{d+1} mapdim^{(i,d+1)}\ h$$
$$= \quad \{ \text{ Definition of the } \bigcirc \text{-notation and of } mapdim \ \}$$
$$chdim^{(k-1,d+1)} \circ map^{d}\ h \circ chdim^{(k-1,d+1)} \circ \bigcirc_{i=k}^{d+1} mapdim^{(i,d+1)}\ h$$
$$= \quad \{ \text{ Induction hypothesis } \}$$
$$chdim^{(k-1,d+1)} \circ map^{d}\ h \circ chdim^{(k-1,d+1)} \circ \bigcirc_{i=k}^{d} chdim^{(i,i+1)} \circ \bigcirc_{i=k}^{d+1}\left(map^{d}\ h \circ chdim^{(i,d+1)}\right)$$
$$= \quad \{ \text{ Promote } \bigcirc_{i=k}^{d-1} chdim^{(i,i+1)} \text{ to the left through independent dimensions } \}$$
$$\left(\bigcirc_{i=k}^{d-1} chdim^{(i,i+1)}\right) \circ chdim^{(k-1,d+1)} \circ map^{d}\ h \circ chdim^{(k-1,d+1)} \circ chdim^{(d,d+1)} \circ$$
$$\bigcirc_{i=k}^{d+1}\left(map^{d}\ h \circ chdim^{(i,d+1)}\right)$$
$$= \quad \{ chdim^{(k-1,d+1)} \circ chdim^{(d,d+1)} = chdim^{(k-1,d)} \circ chdim^{(k-1,d+1)};\ map^{d}\ h \text{ to right } \}$$
$$\left(\bigcirc_{i=k}^{d-1} chdim^{(i,i+1)}\right) \circ chdim^{(k-1,d+1)} \circ chdim^{(k-1,d)} \circ map^{d}\ h \circ chdim^{(k-1,d+1)} \circ$$
$$\bigcirc_{i=k}^{d+1}\left(map^{d}\ h \circ chdim^{(i,d+1)}\right)$$
$$= \quad \{ \text{ Absorbing right } \}$$
$$\left(\bigcirc_{i=k}^{d-1} chdim^{(i,i+1)}\right) \circ chdim^{(k-1,d+1)} \circ chdim^{(k-1,d)} \circ \bigcirc_{i=k-1}^{d+1}\left(map^{d}\ h \circ chdim^{(i,d+1)}\right)$$
$$= \quad \{ chdim^{(k-1,d+1)} \circ chdim^{(k-1,d)} = chdim^{(d,d+1)} \circ chdim^{(k-1,d+1)};\ \text{absorbing } \}$$
$$\left(\bigcirc_{i=k}^{d} chdim^{(i,i+1)}\right) \circ chdim^{(k-1,d+1)} \circ \bigcirc_{i=k-1}^{d+1}\left(map^{d}\ h \circ chdim^{(i,d+1)}\right)$$
$$= \quad \{ chdim^{(s,s+1)} \circ chdim^{(k-1,s+1)} = chdim^{(k-1,s)} \circ chdim^{(s,s+1)}, k \leq s \leq d \ \}$$
$$\bigcirc_{i=k-1}^{d} chdim^{(i,i+1)} \circ \bigcirc_{i=k-1}^{d+1}\left(map^{d}\ h \circ chdim^{(i,d+1)}\right)$$
$\square$.