Formal Derivation of Divide-and-Conquer Programs: A Case Study in the Multidimensional FFT's

Sergei Gorlatch and Holger Bischof

University of Passau, D–94030 Passau, Germany
Email: gorlatch@fmi.uni-passau.de, Fax: +49.851.509.3092

Abstract

This paper reports a case study in the development of parallel programs in the Bird-Meertens formalism (BMF), starting from divide-and-conquer algorithm specifications. The contribution of the paper is two-fold: (1) we classify divide-and-conquer algorithms and formally derive a parameterized family of parallel implementations for an important subclass of divide-and-conquer, called DH (distributable homomorphisms); (2) we systematically adjust the mathematical specification of the Fast Fourier Transform (FFT) to the DH format and thereby obtain a generic SPMD program, well suited for implementation under MPI. The target program includes the efficient FFT solutions used in practice – the binary-exchange and the 2D- and 3D-transpose implementations – as its special cases.

1. Motivation

Formal methods offer a way to cope with the problems of programming parallel machines. They can be viewed as an alternative to the current practice of low-level programming, when the correctness and performance of programs strongly depend on the programmer's experience and require a cumbersome process of debugging and testing, followed by the performance monitoring and tuning to a particular architecture.

We follow the general approach of transformational program derivation from an intuitively clear and correct specification via exclusive use of semantically sound rules in a suitable formalism. Our formal base is the *Bird-Meertens formalism* (BMF), where higher-order functions on lists are used for specification, and equational reasoning is used for program derivation. More specifically, we take the skeleton-based approach [4] which captures the ubiquitous patterns of parallelism in a mathematically rigorous, higher–order way.

The transformational, skeleton-based approach splits the task of parallel programming between two groups of experts:

- The *system experts*, first, identify a set of parallel skeletons which adequately express non-trivial applications and, second, develop provably correct, portable parallel skeleton implementations with a predictable performance. This work requires a deep knowledge of both the design formalism and the target parallel software.
- The *application experts* should be shielded from both the parallel machine details and the underlying formalism (in our case BMF). Their knowledge of the application domain is used to do non-trivial changes and optimizations in the process of problem specification and the adjustment of the specification to the available set of skeletons.

In the paper, we present a case study, which demonstrates this rôle distribution in the case of divide-and-conquer algorithms. On the system side, we classify algorithms, which are based on the divide-and-conquer strategy, and identify a subclass of divide-and-conquer functions, called DH (Distributable Homomorphisms). For this class, a family of parallel implementations is formally derived and expressed as a generic SPMD program in the MPI style. On the application side, we consider the mathematical specification of the Fast Fourier Transform (FFT) as an example. Using its mathematical properties, the specification is formally adjusted to the DH format, which immediately provides an efficient parallel implementation of FFT.

The power of the approach is demonstrated by the fact that we formally derive a parameterized family of parallel implementations for a class of problems, whose specialization in case of FFT yields the efficient parallel solutions used in practice.

The paper is structured as follows. Section 2 introduces the mathematical specification of our case study, FFT, and the formal notation of BMF. The latter is then used in Section 3 for representation and general classification of divide-and-conquer algorithms. Section 4 revisits the FFT specification and specifies it in BMF. In Section 5, a subclass of divide-and-conquer, DH, is introduced and a generic parallel implementation for that subclass is derived formally. Section 6 proceeds with the FFT case study: we systematically adjust its specification to the DH format and obtain a generic parallel program for FFT, whose special cases are the implementations used in practice. In Section 7, we compare to related work and outline our future research. Section 8 (Appendix) contains formal proofs of the claims made in the paper.

2. Case Study and Formal Notation

In this section, we briefly introduce our case study as a mathematical specification and outline the notation of BMF used for program derivation.

2.1. Fast Fourier Transform

We distinguish in the sequel the general considerations from their illustration for the FFT example by inserting the latter between brackets EXAMPLE $-\ \Diamond$.

EXAMPLE 1 (MATH. SPECIFICATION OF FFT) Consider a list $x = [x_0, \ldots, x_{n-1}]$ of length $n = 2^l$. The Fast Fourier Transform function, which we denote *FFT*, yields for x a list of the same length, whose i-th element is defined as follows (see [10]), with $u = [x_0, x_2, \ldots, x_{n-2}]$ and $v = [x_1, x_3, \ldots, x_{n-1}]$:

$$
\begin{aligned}
\left(FFT\ x \right)_i &= \sum_{k=0}^{n-1} x_k \omega_n^{ki} \\
&= \sum_{k=0}^{n/2-1} x_{2k}\, \omega_{n/2}^{ki} + \omega_n^i \sum_{k=0}^{n/2-1} x_{2k+1}\, \omega_{n/2}^{ki} \qquad (1) \\
&= (FFT\ u)_{i \bmod (n/2)} + \omega_n^i\, (FFT\ v)_{i \bmod (n/2)} \\
&= \begin{cases} (FFT\ u)_i + \omega_n^i\, (FFT\ v)_i & , i < n/2 \\ (FFT\ u)_{i-n/2} - \omega_n^{i-n/2}\, (FFT\ v)_{i-n/2} & , i \geq n/2 \end{cases}
\end{aligned}
$$

where ω_n is the primitive n^{th} root of unity in the complex plane, i.e. $\omega_n = e^{2\pi\sqrt{-1}/n}$, and $i = 0, \ldots, n-1$. This is a typical divide-and-conquer recursion used in many algorithms. $\Diamond$

Our goal is to derive an efficient parallel implementation for specifications similar to (1). We introduce a formal notation and then demonstrate how the divide-and-conquer strategy can be cast in it.

2.2. Bird-Meertens Formalism

In the Bird-Meertens formalism (BMF) [3, 14], higher-order functions are defined on arbitrary composite types (lists, trees, arrays, etc.), and equational reasoning is used for deriving programs by transformation. We restrict ourselves to non-empty lists and use the following notation (for brevity, definitions are informal):

$[\alpha]_k$	the type of lists of length k whose elements are of type α;
$+\!\!+$	list concatenation;
$\circ$	backward functional composition;
map f	*map* of an unary function f : *map* $f\ [x_1, \ldots, x_n] \stackrel{\text{def}}{=} [\, f\ x_1, \ldots, f\ x_n \,]$;
red $(\odot)$	*reduce* with a binary associative operation $\odot : \alpha \to \alpha \to \alpha$, *red* $(\odot)\, [x_1, \ldots, x_n] \stackrel{\text{def}}{=} x_1 \odot x_2 \odot \cdots \odot x_n$;
zip $(\odot)$	combines elements of two lists of equal length with operation $\odot$, *zip* $(\odot)\ (\, [x_1, \ldots, x_n]\, , [y_1, \ldots, y_n]\,) \stackrel{\text{def}}{=} [\, (x_1 \odot y_1), \ldots, (x_n \odot y_n)\,]$.

Function application is denoted by juxtaposition, is tightest binding, and associates to the left. We will also use the explicit function application, @, with the fully curried version of *zip* which we denote $\widehat{zip}$:

$$
\widehat{zip}\, (@)\, [f_1, \ldots, f_k]\, [x_1, \ldots, x_k] \;=\; [f_1\, x_1, \ldots, f_k\, x_k] \qquad (2)
$$

From the Backus' FP notation [2], we borrow the following two combinators:

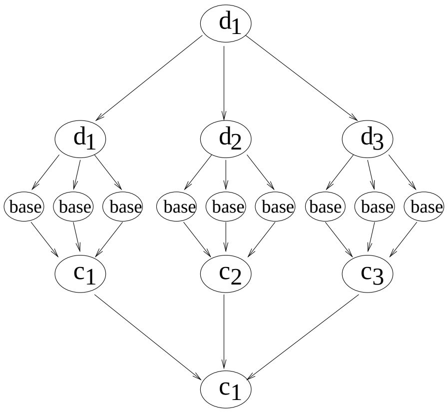
construction: $< f_1, \ldots, f_n >\ x \stackrel{\text{def}}{=} (f_1\, x, \ldots, f_n\, x)$

and also the *conditional* defined as follows:

$(p \mapsto A;\, B)\, x \stackrel{\text{def}}{=}$ `if` p `then` $A\, x$ `else` $B\, x$.

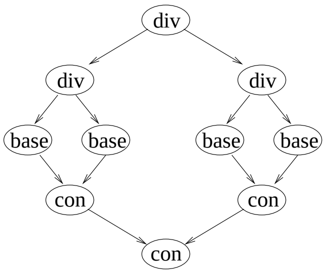
The Bird-Meertens formalism was introduced as a transformational framework for the design of (at first sequential) algorithms and programs. We use BMF for representing both initial specifications and the parallel target implementations as expressions of the formalism. The equational theory of BMF is used for transforming specifications into implementations. The exclusive use of semantically sound BMF rules guarantees the correctness of the target program with respect to the initial specification. Our development process stops when the functional representation has a direct imperative SPMD implementation.

For examples of BMF equational rules and their use for program derivation, see Section 8.



$$
\begin{array}{l}
SDC\ k\ l\ div\ con\ base\ =\ SDC_1\ k\ l\ div\ con\ base, \\
\quad \text{where} \\
\quad SDC_i\ k\ l\ div\ con\ base\ = \\
\qquad (l=0)\ \mapsto\ base\,; \\
\qquad c_i\ \circ\ \left(\ \widehat{zip}\,(@)\left[\,SDC_j\ k\ (l{-}1)\ div\ con\ base\,|\,j=1..k\,\right]\ \right)\ \circ\ d_i. \\
\quad div=(d_1,\ldots,d_k);\ con=(c_1,\ldots,c_k); \\
\quad d_i:\alpha\rightarrow[\alpha]_k;\ c_i:[\alpha]_k\rightarrow\alpha;\ i=1,\ldots,k.
\end{array}
$$

Figure 1. SDC: Static Divide-and-Conquer for $k=3, l=2$.



$$
\begin{array}{l}
DC\ l\ div\ con\ base\ = \\
\quad (l=0)\ \mapsto\ base\,; \\
\quad con\ \circ\ map\,(DC\ (l{-}1)\ div\ con\ base)\ \circ\ div.
\end{array} \tag{3}
$$

$$div:\alpha\rightarrow[\alpha];\ con:[\alpha]\rightarrow\alpha.$$

Figure 2. DC: binary, symmetric SDC

3. Divide-and-Conquer in BMF

Let us use the introduced notation for the specification of divide-and-conquer. We will also give a brief classification of divide-and-conquer algorithms we are going to study.

3.1. Static Divide and Conquer

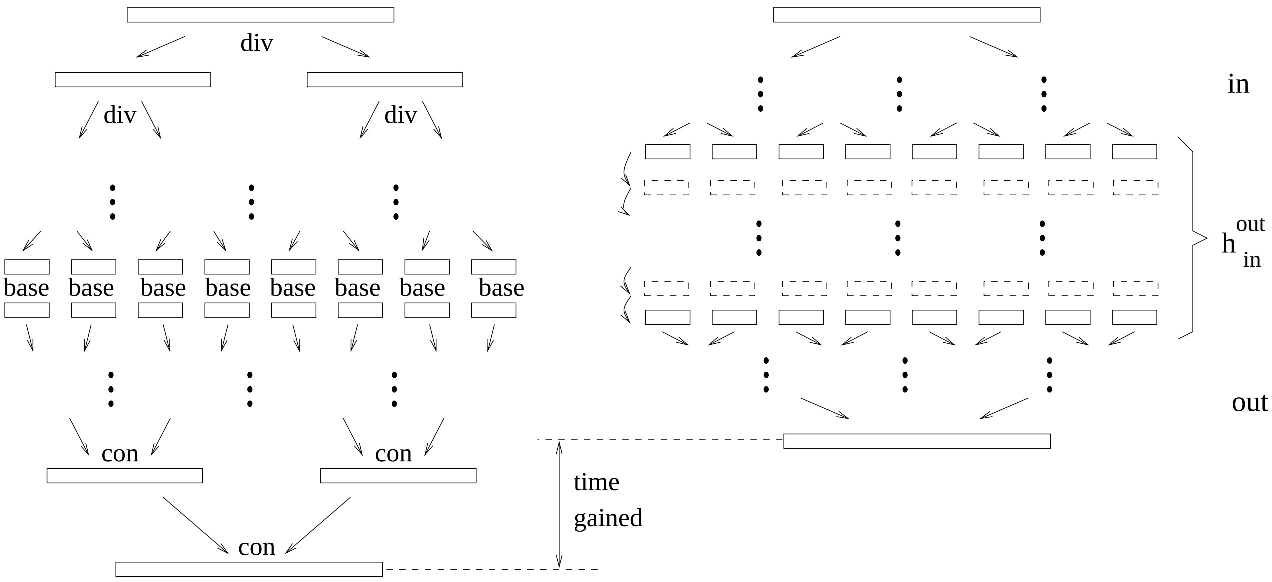
We restrict ourselves to the class of divide-and-conquer algorithms, whose control structure can be determined at compile time; we call them *SDC* – algorithms, for Static Divide-and-Conquer. We represent this form of divide-and-conquer as the higher-order function *SDC*, illustrated in Figure 1, whose parameters are:

- k – the division degree: the number of parts in which the original problem is divided;
- l – the recursion depth: the number of divide (and thus also conquer) steps; it may not be known at compile time and remains a parameter but it ensures that all branches in the call tree have the same depth.
- $div=(d_1,\ldots,d_k)$ – the tuple of k divide functions where k is the division degree;
- $con=(c_1,\ldots,c_k)$ – the tuple of k conquer functions;
- *base* – the function applied in the base case.

For a definition and illustration of *SDC*, see Figure 1, where $\widehat{zip}$ is used analogous to (2), but is supplied with only the first of its two list arguments. This list argument is expressed in the Haskell-like notation: $[x_i|i=1..k]=[x_1,\ldots,x_k]$. To avoid the possible type inference problems, particular versions of the *SDC*-skeleton, such as binary, ternary, etc., may be more preferable in practice than the general format.

The most well-known and widely used divide-and-conquer scheme is the binary, symmetric version of the *SDC* format, with $k=2$ and $d_1=d_2=div$, $c_1=c_2=con$. In the sequel, we consider for simplicity this schema which is defined as the higher-order function *DC* on lists in Figure 2; note the changed types of functions $div:\alpha\rightarrow[\alpha]$ and $con:[\alpha]\rightarrow\alpha$ comparing to the *SDC* format. [1]

[1] Note that all considerations in this section, in particular Theorem 2, can be easily reformulated for the general static case with an arbitrary fixed division degree.



$$\bigcirc_{i=0}^{l-1}(map^i\ con)\ \circ\ (map^l\ base)\ \circ\ \bigcirc_{i=0}^{l-1}(map^{l-i-1}\ div) \qquad \Rightarrow \qquad out\ \circ\ h_{in}^{out}\ \circ\ in$$

Figure 3. Direct iterative implementation (left) and Distributed implementation (right).

3.2. Direct Implementation

The BMF functional *map* in (3) points to the main source of parallelism in DC-algorithms, namely the independent computations on the two parts after division; we used similar representation in [9]. Our goal is to develop an efficient parallel program from a DC specification.

Let us first look at what kind of parallelism is hidden in the recursive specification (3) and what one could expect from its implementation. The easiest way to see this is to simply unfold the recursion.

We first explain the unfolding intuitively and then formulate the corresponding relation more precisely. During the divide phase, we first apply function *div* to the input, then *div* is applied further to both result parts simultaneously, and so on, until after l steps we arrive at the base case. The nesting of the lists is growing with each divide step; to apply function f to the elements of a list of nesting depth 2, we use the functional *map* as follows:

$$map\ (map\, f)\ [\,[1,2],[3,4]\,]\ =\ [\,[f\,1,\, f\,2],\,[f\,3,\, f\,4]\,]$$

By denoting $map^l\ div \stackrel{\mathrm{def}}{=} map\ (map^{l-1}\ div)$, the entire divide phase rewrites as the following multiple (sequential) composition:

$$(map^{l-1}\ div) \circ \cdots \circ (map\ div) \circ div \qquad (4)$$

We introduce a special notation for a multiple functional composition, $\bigcirc_{i=0}^{l-1}$, which has the same meaning for $\circ$ as $\sum_{i=0}^{l-1}$ has for $+$. Then (4) is expressed by $\bigcirc_{i=0}^{l-1}(map^{l-i-1}\ div)$, where $map^0\ div \stackrel{\mathrm{def}}{=} div$. Note that $\circ$ is backward, so expression (4) is executed from right to left. The result of the divide phase is a nested list of depth l, so it remains to compute $map^l\ base$ and then proceed with the conquer phase. The unfolding of (3) thus, not very surprisingly, yields a composition of the three iterative phases – divide, base case and conquer:

$$IDC\ l\ div\ con\ base\ =$$
$$\bigcirc_{i=0}^{l-1}(map^i\ con)\ \circ\ (map^l\ base)\ \circ\ \bigcirc_{i=0}^{l-1}(map^{l-i-1}\ div) \qquad (5)$$

We call (5) the *iterative* representation of divide-and-conquer (*IDC*); it is illustrated in the left part of Figure 3.

Theorem 2 *Recursive representation* (3) *and iterative representation* (5) *are semantically equivalent.*

Proof See Appendix. □

Representation (5) shows the nature of divide-and-conquer more explicitly than (3) does. In particular, the number of sequential composition steps equals the depth of recursion, so the parallel time complexity is proportional to l. In case of $FFT\, x$, $l = \log |x|$, where $|x|$ denotes the length of list x.

However, there are problems with direct parallel implementation of (5), see Figure 3:

- Both *div* and *con* are list functions requiring communication between the tree nodes; the communication time is linear in the problem size.

- The breadth of the graph in Figure 3 and thus the amount of available parallelism is non-constant, so the load must be additionally balanced, either at compile time or at run time.

- The *DC*-specification both takes and yields a monolithic list, which may induce additional data redistribution costs when the algorithm is used as a part of a big parallel/distributed application.

We aim at an implementation, which is directly and efficiently implementable on multicomputers with distributed memory. In [9], we have pursued this goal in the general case by harnessing the divide phase according to the available number of processors. In this paper, we take a different approach.

3.3. Distributed Implementation

To overcome the problems of the direct implementation mentioned above, we aim instead at a so-called *distributed implementation* of divide-and-conquer. The transition from the direct implementation (5) to a distributed implementation is illustrated in Figure 3 on the right: we extract the data arrangement (without computation) from the divide stage of the algorithm and express it as function *in*; similarly, the closing data rearrangement is extracted from the conquer stage and represented by function *out*.

A distributed implementation thus consists of three stages: it starts with *in* and ends with *out*; the central part, h_{in}^{out}, called the *distributed phase* of divide-and-conquer, takes data arranged by *in* and yields data which can be rearranged afterwards by function *out*. Function h_{in}^{out} comprises the "useful" work of the divide-and-conquer algorithm. The distributed implementation as a whole is efficient if the work is mostly done in the distributed phase, with a constant breadth of parallelism; in that case, time can be gained comparing to the direct implementation. We are usually looking for a distributed phase of a given divide-and-conquer function in the form of a sequence of parallel stages [6]; such form is well suited for an SPMD implementation.

The problem of finding a distributed implementation can be formulated in two ways:

- For a given divide-and-conquer specification of a function h on monolithic data structures, find an efficient distributed implementation, i.e. the distributed phase h_{in}^{out} together with the corresponding *in* and *out*.

- For given functions *in* and/or *out*, which specify the available input and output data arrangements, find an optimal distributed phase of the algorithm.

3.4. Separated and Nested DC

The separation of data rearrangements from the divide-and-conquer is sometimes simplified by the specific format of an algorithm. As observed elsewhere, in many practical divide-and-conquer algorithms either divide or conquer stage is rudimentary in that it just describes a special data representation and does no "useful" work. Separation of the divide or the conquer phase leads to so-called pre- and postmorphisms or top-down and bottom-up algorithms respectively [1, 12].

Following our *DC* acronym, we call such one-sided schemes D– and C–algorithms, i.e. "Divide (without Conquer)" and "Conquer (without Divide)", correspondingly. For example, list homomorphisms in BMF [3] are computable by C–algorithms: their divide phase can be separated and represented by function *in* that transforms an input lists into a list of lists.

The divide-and-conquer specifications may also have a nested control structure. For example, the problem of parsing many-bracket languages is a so-called nested almost-homomorphism [8], i.e. a C–algorithm, whose *con* is again a C–function; we denote this class by $C(C)$. Another well-known example is bitonic sort, which belongs to the class $C(D)$; see [1, 12] for details.

4. Case Study: Revisited

Example 3 (FFT as divide-and-conquer) The mathematical specification (1) of FFT can be directly represented in the divide-and-conquer format (3) as follows:

$$FFT\ x \;=\; DC\ l\ div\ con\ base\ x\,,$$

where

$$\begin{aligned}
l &= \log|x|,\\
base &= id - \text{the identity function},\\
div\,[x_0,\ldots,x_{n-1}] & \\
&= [\,[x_0,x_2,\ldots,x_{n-2}]\,,\,[x_1,x_3,\ldots,x_{n-1}]\,],\\
con\,[u,v] &= zip\,(+)\ (u, zip\,(\cdot)\,(\Omega,v))\ +\!\!+\\
&\qquad zip\,(-)\ (u, zip\,(\cdot)\,(\Omega,v))\\
\Omega &= [\omega_n^0,\omega_n^1,\ldots,\omega_n^{n/2-1}],\\
\omega_n &- \text{primitive } n^{th} \text{ root of unity.}
\end{aligned}$$

Note that function *FFT* can be specified as a $C-$algorithm, which takes data arranged cyclically. $\Diamond$

5. Deriving Distributed Implementation

Now we address the question: for which divide-and-conquer algorithms can we systematically derive an efficient distributed implementation with a balanced amount of parallelism like on the right side of Figure 3. We call such algorithms *distributable*.

Our approach is to use nested lists and to restrict the general format given by (3).

5.1. Nested Lists

The type of d-nested lists, with the same number m of sublists at all levels is denoted by

$$[\alpha]^d_m \;=\; [\ldots[\alpha]_m\ldots]_m$$

If we have a list of lists, whose type is $[\alpha]^2_m$, then the usual *map* functional allows to apply a list function $h:[\alpha]\to[\alpha]$ to each innermost list or, in the matrix form, to each row of the matrix:

$$map\ h\ \begin{bmatrix}[a_{11} & \ldots & a_{1m}],\\ [a_{21} & \ldots & a_{2m}],\\ \vdots & & \vdots\\ [a_{m1} & \ldots & a_{mm}]\end{bmatrix} = \begin{bmatrix}h[a_{11} & \ldots & a_{1m}],\\ h[a_{21} & \ldots & a_{2m}],\\ \vdots & & \vdots\\ h[a_{m1} & \ldots & a_{mm}]\end{bmatrix} \quad (6)$$

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:

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

Note that the usual 2D matrix transposition is a special case of the *chdim* function: $transpose = chdim^{(1,2)}$. We use function *chdim* for defining a generalized version of *map*, functional *mapdim*, for lists of type $[\alpha]^d$, function $h:[\alpha]\to[\alpha]$ and $1\le i\le d$:

$$\begin{aligned}
&mapdim^{(i,d)}\ h \stackrel{\text{def}}{=}\\
&\qquad chdim^{(i,d)}\ \circ\ map^{(d-1)}\,h\ \circ\ chdim^{(i,d)} \qquad (7)
\end{aligned}$$

Note that $mapdim^{(d,d)} = map^{(d-1)}$, so (6) effectively illustrates $mapdim^{(2,2)} \;=\; map$.

Mapping along the columns of a two-dimensional list is now captured by $mapdim^{(1,2)}$:

$$\begin{aligned}
mapdim^{(1,2)}\ \ h\ \begin{bmatrix}[a_{11} & \ldots & a_{1m}],\\ [a_{21} & \ldots & a_{2m}],\\ \vdots & & \vdots\\ [a_{m1} & \ldots & a_{mm}]\end{bmatrix} &=\\
transpose \left(\begin{bmatrix}h[a_{11} & \ldots & a_{m1}],\\ h[a_{12} & \ldots & a_{m2}],\\ \vdots & & \vdots\\ h[a_{1m} & \ldots & a_{mm}]\end{bmatrix}\right) & \qquad (8)
\end{aligned}$$

We have introduced here only the notions which are necessary for our case study; for more technical details see the Appendix.

5.2. Distributable Homomorphisms

Let us now restrict the general *DC* format (3) in order to make divide-and-conquer functions distributable, i.e. to enable their distributed implementation. From now on, we assume that lists' length is a power of 2 (they are also called *powerlists* [11]), function *div* splits the input list in the middle, without any additional rearrangements, and *con* is a componentwise function. These restrictions are captured in the class DH (*distributable homomorphisms*), first introduced in [7]; DH is a subclass of list homomorphisms [3].

Definition 4 For two binary operations, $\oplus$ and $\otimes$, a distributable homomorphism is function $(\oplus \updownarrow \otimes)$: $[\alpha]\to[\alpha]$, such that for arbitrary lists x and y of equal lengths holds:

$$\begin{aligned}
(\oplus\updownarrow\otimes)\ (x +\!\!+ y) \;&=\; zip\,(\oplus)\,(u,v)\ +\!\!+\ zip\,(\otimes)\,(u,v) \quad (9)\\
\text{where}\ \ u \;=\; (\oplus\updownarrow\otimes)\,x\,,\ \ v \;&=\; (\oplus\updownarrow\otimes)\,y
\end{aligned}$$

$\square$

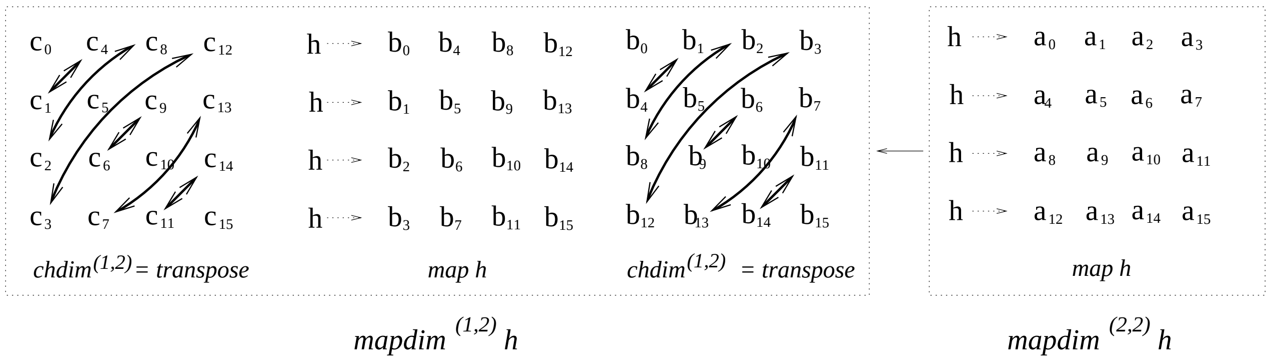


Figure 4. Implementation (10) **in the two-dimensional case,** $d = 1$**.**

To define a DH function uniquely, we must also provide its value on a singleton list; usually $(\oplus \updownarrow \otimes)[a] \stackrel{\text{def}}{=} [a]$. Figure 5 illustrates, how a distributable homomorphism h is computed on a concatenation of two lists; dashed arrows denote replication of the partial results:

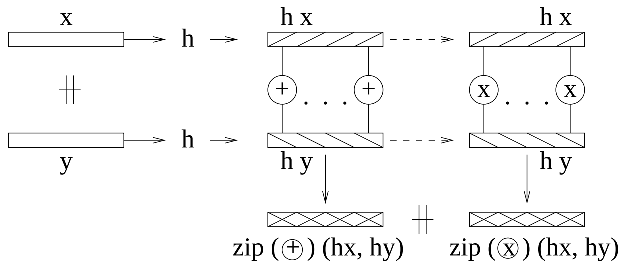


Figure 5. Distributable homomorphism h**: computing** $h(x + y)$

According to our classification in Subsection 3.4, every DH-function can be clearly specified as a C–algorithm.

5.3. Distributed Implementation

To study the parallel implementation of DH, let us use nested lists, restricting ourselves to "quadratic" 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 $m = \sqrt{|x|}$.

We denote $flat^d$ the left inverse of $nest^d$: it flattens a $(d+1)$-dimensional list. It is obvious that $flat^1 = red(+\!\!+)$.

The following central proposition shows how the class DH can be implemented in parallel.

THEOREM 5 (DISTRIBUTED IMPLEMENTATION OF DH) *Every distributable homomorphism h on a list of length 2^l can be computed for each $d = 1, \ldots, l-1$ as follows:*

$$h \;=\; flat^d \circ \left(\bigcirc_{i=1}^{d+1} mapdim^{(i,d+1)}\, h \right) \circ nest^d \qquad (10)$$

PROOF See Appendix. □

Theorem 5 provides a family of parallel implementations for a DH, one for each $d = 1, \ldots, l-1$. All these implementations are distributed implementations according to the right half of Figure 3:

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

where

$$h_{in}^{out} \;=\; \bigcirc_{i=1}^{d+1} mapdim^{(i,d+1)}\, h\,, \qquad (11)$$

$$in \;=\; nest^d,$$

$$out \;=\; flat^d$$

Functions in, out and h_{in}^{out} have d as a parameter, whose value will usually be clear from the context and thus omitted.

Let us look at implementations which are two extreme special cases of (11). The first one, for $d = 1$, prescribes the 2D data arrangement, i.e. a matrix, on which two *maps* presented by (6) and (8) are computed, with two transpositions of the form $transpose = chdim^{(1,2)}$: one between *maps* and the other afterwards; see Figure 4. Note that for $i = 2$, both *chdims* disappear since $chdim^{(2,2)} = id$.

The other extreme special case is $d = l-1$: the input list is transformed in a logical l-dimensional hypercube and 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.

Interestingly enough, the first special case is known in the literature on FFT as the 2D-transpose schema and the second as the binary-exchange schema. Besides these two schemata, Theorem 5 yields the whole spectrum of algorithms between them, including also the 3D algorithm described in [10]. Our family of algorithms work for the whole class of DH functions, with FFT being just one representative of the class.

5.4. Optimization

Let us consider function h_{in}^{out} of (11). Due to (7), there are two *chdims* between every pair of *maps* in the multiple composition. The change of dimension after each *map* does not influence the computations performed afterwards, so we can push all such data rearrangements towards the end of the expression.

THEOREM 6 (COMMUNICATIONS REARRANGEMENT)

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

PROOF See Appendix.

□

The result expression for h_{in}^{out} is a composition of two parts, the second of which, $\bigcirc_{i=1}^{d} chdim^{(i,i+1)}$, does the shift of dimensions to the right: dimension i becomes $i+1$, dimension $d+1$ becomes the first one. If we incorporate that rearrangement into the *out*-function of (11), then the distributed implementation of a distributable homomorphism h reads as follows:

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

where

$$h_{in}^{out} = \bigcirc_{i=1}^{d+1} \left(map^{\,d}\, h \circ chdim^{(i,d+1)} \right), \quad (12)$$

$$in = nest^{d},$$

$$out = flat^{d} \circ \bigcirc_{i=1}^{d} chdim^{(i,i+1)}$$

Functions *in* and *out* specify that the distributed phase h_{in}^{out} in (12), for a given d, takes the data as $(d+1)$-dimensional array and yields the result array of the same dimension, with dimensions shifted to the left.

5.5. Bounded Number of Processors

We address now a practical problem: how can the distributed phase of h, function h_{in}^{out} in (12), be implemented on an arbitrary but bounded number of processors, p.

First we should decide how the $(d+1)$-dimensional data are to be distributed between the processors. We aim at a balanced distribution, such that for an input of size $n = 2^l$, the number of data elements stored as a block in one processor is n/p. One possible solution is as follows. We restrict the dimensionality d and thus also the number of different implementations by $\log p$. For each $d, 1 \le d \le \log p$, processors constitute a virtual d-dimensional structure, with $\sqrt[d]{p}$ processors in each dimension. This structure is just a logical enumeration of processors: we make no assumptions about the processor topology since we aim at an architecture-independent solution. Data constitutes a $(d+1)$-dimensional list, with $n^{1/(d+1)}$ elements at each level. First d dimensions are distributed evenly between processors, the last dimension is not distributed, so each processor stores a $(d+1)$-dimensional block of size: $\frac{\sqrt[d+1]{n}}{\sqrt[d]{p}} \times \cdots \times \frac{\sqrt[d+1]{n}}{\sqrt[d]{p}} \times \sqrt[d+1]{n}$.

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. Case $d = 1$ corresponds as before to a matrix, but now its rows are aggregated in p consecutive blocks, stored in a one-dimensional array of p processors; the second dimension is not distributed. The three-dimensional case, $d = 2$, is shown in Figure 6, borrowed from [10]. Case $d = \log p$ corresponds to the p-processor hypercube.

According to (12), the generic distributed phase proceeds in $(d+1)$ sequential stages, with computation and communication clearly separated within one stage. The first part of each stage, expressed by *chdim*, is a transposition of two dimensions. It requires a personalized all-to-all communication of processors (which can be implemented by, e.g., the MPI primitive All-To-All [15]), followed by an intra-processor rearrangement. The second part of a stage computes the function along the last dimension. Since the last dimension is not distributed, such computations on different processors are independent. On p processors, the potential parallelism of (12) is restricted: each processor executes its $\sqrt[d+1]{n^d}/p$ pieces of computation of h sequentially, although they are independent of each other.

Figure 6 illustrates the transition between the first two of altogether three stages in the three-dimensional special case, $d = 2$, on p processors. In the left part of the figure, function h is computed in all processors

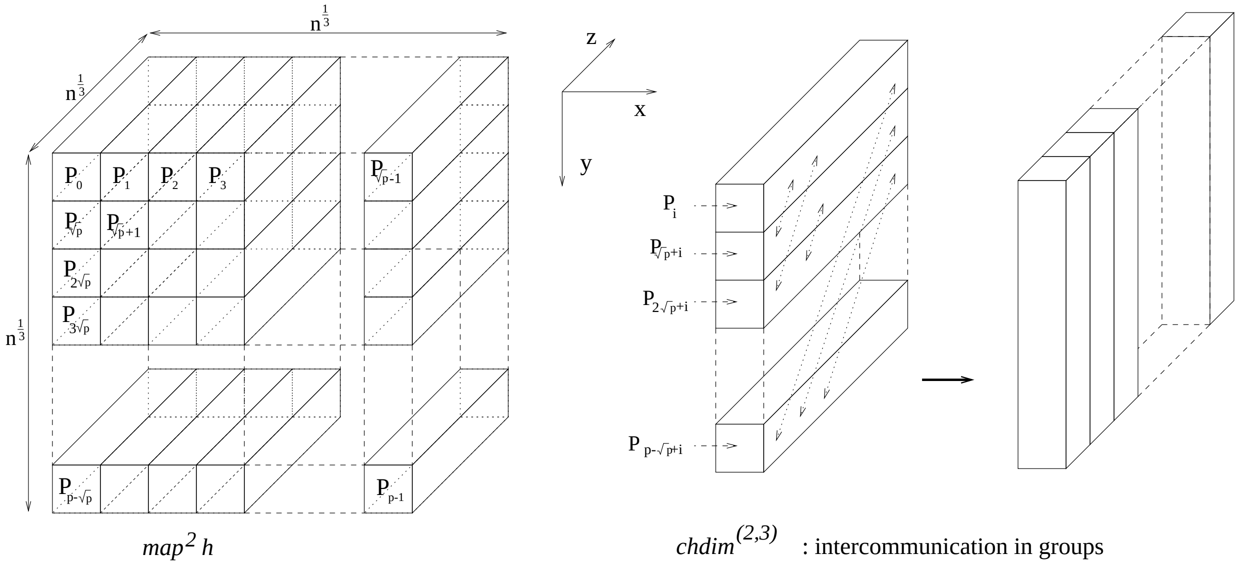


Figure 6. Three-dimensional case, $d = 2$: the transition between the first and the second stage.

simultaneously along the last dimension, z (the first stage). The right part of the figure illustrates the beginning of the second stage which is associated with dimension y: it starts by changing dimensions z and y, which is done by the all-to-all communication within each group of processors located in one plane $y-z$. The stage proceeds then further by computing function h along y in all processors simultaneously.

This distributed implementation can be directly represented as the following pseudo-MPI program:

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

Note that the SPMD program structure corresponds directly to the structure of the BMF-implementation (12). Functions $in = nest^d$ and $out = flat^d \circ \bigcirc_{i=1}^{d+1} chdim^{(d+1,i)}$ are included as annotations of the input and output data distributions; the output distribution is shifted to the left. The `All-to-all` statement with the `Local-transpose` afterwards, implement $chdim^{(i,d+1)}$ of (12). Communication happens in the groups which are arranged by statement `group`: it is an analogue of communicator in MPI which composes the processors into groups depending on the dimension d and iteration i. Function `Map` computes function h along the undistributed dimension for the whole block.

6. Application to FFT

To use the generic program (12) on p processors for a particular divide-and-conquer function, we must adjust that function to the DH format. Let us do that for the FFT example. We make the steps of the adjustment process explicit, in order to outline a possible systematic way of adjustment.

Step 1. First we need to change function div from an odd-even division as in Example 3 to splitting a list in the middle as required by the definition of DH. This can be reached by inverting the input list with function inv [1]: an element with index i in list x has the reversal of bit string i as its index in $inv\,x$. This initial arrangement of the FFT input is also used in practice.

The divide-and-conquer representation of the "inverted" FFT, denoted by $\widetilde{FFT}$, is already much nearer to the desired DH format:

$$
\begin{aligned}
\widetilde{FFT} &= DC\ l\ div\ con\ id, \\
\text{where}\quad div(x +\!\!+ y) &= [x, y] \\
con[u, v] &= zip\,(+)\ (u, zip(\cdot)(\Omega, v))\ +\!\!+ \\
&\ zip\,(-)\ (u, zip(\cdot)(\Omega, v))
\end{aligned}
$$

Actually, $\widetilde{FFT}$ is a C–algorithm since its div just rearranges data.

Step 2. Let us now cast function $\widetilde{FFT}$ in the DH-format (9), denoting $u = \widetilde{FFT}\,x$, $v = \widetilde{FFT}\,y$:

$$\left(\widetilde{FFT}(x +\!\!+ y)\right)_{(i)} = (con\,[u,v])_{(i)}$$
$$= \left(zip\,(+)\,(u, zip(\cdot)(\Omega, v)) +\!\!+ zip\,(-)\,(u, zip(\cdot)(\Omega, v))\right)_{(i)}$$
$$= \begin{cases} u_{(i)} + \omega_{2m}^{i} \cdot v_{(i)}, \text{ if } i < m \overset{\text{def}}{=} |u| = |x| \\ u_{(i-m)} - \omega_{2m}^{i-m} \cdot v_{(i-m)}, \text{ otherwise} \end{cases}$$

We have almost arrived at the desired component-wise computation on u and v, the only remaining problem being the coefficients ω_{2m}^{i} and ω_{2m}^{i-m}. Observing that:

$$\begin{aligned} i &= (j \bmod m) \quad , \text{if } i < m \\ (i-m) &= (j \bmod m) \quad , \text{if } i \geq m, \end{aligned}$$

we obtain for $\left(\widetilde{FFT}(x +\!\!+ y)\right)_{(i)}$:

$$\begin{cases} u_{(i)} + \omega_{2m}^{(j \bmod m)} \cdot v_{(i)}, \text{ if } i < m \\ u_{(i-m)} - \omega_{2m}^{(j \bmod m)} \cdot v_{(i-m)}, \text{ otherwise} \end{cases}$$

Thus FFT is not exactly DH, but rather a so-called "almost-DH" [7]. The standard method of dealing with such functions is to tuple them together with the necessary auxiliary functions; in our case, these are functions J and M yielding the values of j and m correspondingly. The new function, $\widetilde{FFT3}$, takes and yields a list of triples. The initial expansion of the input list into a list of triples is done by function *tri*:

$$tri\,[a_0, \ldots, a_{n-1}] = \left[\begin{pmatrix} a_0 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} a_1 \\ 1 \\ 1 \end{pmatrix}, \ldots, \begin{pmatrix} a_{n-1} \\ n-1 \\ 1 \end{pmatrix} \right]$$

To compute the original function $\widetilde{FFT}$, we should just take the first component of each triple in the result list, which is done by the projection function π_1:

$$\widetilde{FFT} = (map\,\pi_1) \circ \widetilde{FFT3} \circ tri \qquad (13)$$

Auxiliary functions J and M are themselves DH:

$$\begin{aligned} J(x +\!\!+ y) &= zip\,(\pi_1)\,(Jx, Jy) +\!\!+ zip\,(\pi_2)\,(Jx, Jy) \\ M\,(x +\!\!+ y) &= zip\,(\oslash)\,(M\,x, M\,y) +\!\!+ zip\,(\ominus)\,(M\,x, M\,y) \end{aligned}$$

where π_1, π_2 are projection functions and $a \oslash b = 2a$, $a \ominus b = 2b$.

Thus the tupled function is DH, $\widetilde{FFT3} = \oplus \updownarrow \otimes$, with the following operations $\oplus$ and $\otimes$:

$$\begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix} \oplus \begin{pmatrix} a_2 \\ b_2 \\ c_2 \end{pmatrix} = \begin{pmatrix} a_1 + \omega_{2c_1}^{(b_1 \bmod c_1)} \cdot a_2 \\ b_1 \\ 2c_1 \end{pmatrix},$$

$$\begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix} \otimes \begin{pmatrix} a_2 \\ b_2 \\ c_2 \end{pmatrix} = \begin{pmatrix} a_1 - \omega_{2c_2}^{(b_2 \bmod c_2)} \cdot a_2 \\ b_2 \\ 2c_2 \end{pmatrix}$$

Step 3. Now we are ready to use the generic implementation of DH for computing $\widetilde{FFT}$. It remains to substitute into (13) the generic program (12), customized with particular operations for FFT:

$$\begin{aligned} \widetilde{FFT} \;=\; & (map\,\pi_1) \circ flat^{d} \circ \\ & \bigcirc_{i=1}^{d+1} \left((map^{\,d}\,\widetilde{FFT3}) \circ chdim^{(i,d+1)} \right) \circ \\ & nest^{d} \circ tri \end{aligned} \qquad (14)$$

Note that (14) is a generic program for computing function $\widetilde{FFT}$, which depends on the data distribution depth, d. For a given number of processors, p, we can choose one of parallel implementations for d from 1 to $\log p$, as discussed in Section 5.

All three algorithms used for computing FFT in practice [10] – the 2D and 3D transposition and the binary exchange algorithm – are described by the program (14). Their performance evaluation shows an interesting trend: as the dimension 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. Depending on the network characteristics of the particular machine, the most efficient parallel implementation of FFT on that machine can be chosen.

7. Conclusion

We use the BMF functional notation and transformation rules to derive efficient parallel programs for divide-and-conquer algorithms.

The paper demonstrates a systematic, formally based approach to parallel programming, in which the development work is split between the system experts and the application experts as follows:

- On the system side, we use the BMF notation to specify and classify the divide-and-conquer algorithms. For a specific subclass, DH functions, a family of efficient parallel implementations is formally derived by transformation. The result is a higher-order algorithmic skeleton, together with a parameterized distributed implementation. This implementation is provably correct, architecture-independent and has predictable performance.

- On the application side, we demonstrate the use of the derived skeleton for a particular case study. We start with a mathematical specification of FFT, systematically adjust it to the DH skeleton format and subsequently obtain a generic parallel implementation of FFT.

The feasibility of the approach is confirmed by the fact that the obtained implementation includes the FFT solutions used in practice (binary exchange and both 2D and 3D transpose) as its special cases.

Comparing with the verification approaches to parallel programming, we provide a systematic method of deriving efficient parallel implementations, which are correct by construction due to the exclusive use of semantically sound transformation rules of BMF. We use the powerlist structure proposed by Misra [11], but unlike him we provide an explicit iterative solution which is advantageous for the parallel efficiency.

Our target representations of parallel programs are sequences of phases, with clearly decoupled computations and communications like, e.g., programs (12) and (14). Such programs are easy to understand and their performance can usually be easily estimated. They are sometimes viewed as combining both task and data parallelism [5]. Another advantage is that our solutions are architecture-independent and are well suited for implementation by means of collective communication libraries like MPI, which have efficient versions on practically all contemporary parallel and distributed platforms.

The restriction of our approach is in the requirements of the DH format, which at the same time allow us to obtain more practically relevant results than the general approach we pursued in [9].

In our current work, we aim at extending the DH format by relaxing some of its restrictions, and start an experimental study of the derived target programs on a 64-transputer network.

Acknowledgments

Thanks to Christoph Wedler for many technical discussions and to the anonymous referees for their very helpful comments.

This work was partially supported by the DFG (project RecuR2) and DAAD (exchange projects ARC and PROCOPE).

References

[1] K. Achatz and W. Schulte. Architecture independent massive parallelization of divide-and-conquer algorithms. In B. Moeller, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 947, pages 97–127, 1995.

[2] J. W. Backus. Can programming be liberated from the von Neumann style? *Communications of the ACM*, 21:613–641, 1978.

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

[4] M. Cole. Algorithmic skeletons: A structured approach to the management of parallel computation. Ph.D. Thesis. Technical Report CST-56-88, Department of Computer Science, University of Edinburgh, 1988.

[5] P. Dinda et al. The CMU task parallel program suite. Technical Report CMU-CS-94-131, 1994.

[6] S. Gorlatch. Stages and transformations in parallel programming. In M. Kara et al., editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 147–162. IOS Press, 1996.

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

[8] S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In H. Kuchen and D. Swierstra, editors, *Programming languages: Implementation, Logics and Programs. PLILP'96*, Lecture Notes in Computer Science 1140, pages 274–288. Springer-Verlag, 1996.

[9] S. Gorlatch and C. Lengauer. Parallelization of divide-and-conquer in the Bird-Meertens formalism. *Formal Aspects of Computing*, 7(6):663–682, 1995.

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

[11] J. Misra. Powerlist: a structure for parallel recursion. *ACM TOPLAS*, 16(6):1737–1767, 1994.

[12] Z. G. Mou and P. Hudak. An algebraic model for divide-and-conquer algorithms and its parallelism. *Journal of Supercomputing*, 2(3):257–278, 1988.

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

[14] D. Skillicorn. *Foundations of Parallel Programming*. Cambridge U. P., 1994.

[15] D. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20:657–673, 1994.

8. Appendix

8.1. BMF Rules

Some equalities of the Bird-Meertens formalism used as transformation rules:

$$map\ (f \circ g) = map\ f\ \circ\ map\ g \tag{15}$$

$$map\ f\ \circ\ flat = flat\ \circ\ map^2\ f \tag{16}$$

$$red\,(\oplus) \circ flat = red\,(\oplus) \circ map\,(red\,(\oplus)) \tag{17}$$

$$(\oplus \updownarrow \otimes)\ \circ\ flat = red\,(\odot)\ \circ\ map(\oplus \updownarrow \otimes) \tag{18}$$

$$(zip\,\oplus) \updownarrow (zip\,\otimes) = nest\ \circ\ red\,(\odot) \tag{19}$$

In (18), (19): $(\oplus \updownarrow \otimes)\,(x +\!\!+\, y) = (\oplus \updownarrow \otimes)\,x\ \odot\ (\oplus \updownarrow \otimes)\,y$

8.2. Proof of Theorem 2

PROOF Induction on the recursion depth l.
Base case: $l = 0$:

$$
\begin{array}{ll}
& DC\ 0\ \ div\ \ con\ \ base \\
= & \quad \{\ \text{Definition (3)}\ \} \\
& base \\
= & \quad \{\ \text{Definitions of } \bigcirc\text{-notation and of } map^k\ \} \\
& \displaystyle\bigcirc_{i=0}^{-1} (map^i\ con)\ \circ\ (map^0\ base)\ \circ\ \bigcirc_{i=0}^{-1} (map^{l-i-1}\ div) \\
= & \quad \{\ \text{Definition (5)}\ \} \\
& IDC\ 0\ \ div\ \ con\ \ base
\end{array}
$$

Hypothesis: $l = j$.
Induction step: $j \to j+1$:

$$
\begin{array}{ll}
& DC\ (j+1)\ \ div\ \ con\ \ base \\
= & \quad \{\ \text{Definition (3)}\ \} \\
& con\ \circ\ map\,(DC\,j\ \ div\ con\ base)\ \circ\ div \\
= & \quad \{\ \text{Induction hypothesis}\ \} \\
& con\ \circ\ map\,(IDC\,j\ \ div\ con\ base)\ \circ\ div \\
= & \quad \{\ \text{Definition (5) and (15)}\ \} \\
& con\ \circ\ map\,(\displaystyle\bigcirc_{i=1}^{j} (map^i\ con))\ \circ\ map\,(map^j\ base) \circ \\
& \displaystyle\bigcirc_{i=0}^{j-1} (map^{j-i-1}\ div)\ \circ\ div \\
= & \quad \{\ \text{Definitions of } \bigcirc\text{-notation, } map^0 f = f\ \} \\
& \displaystyle\bigcirc_{i=0}^{j} (map^i\ con)\ \circ\ (map^{j+1}\ base)\ \circ\ \bigcirc_{i=0}^{j} (map^{j-i-1}\ div) \\
= & \quad \{\ \text{Definition (5)}\ \} \\
& IDC\ (j+1)\ \ div\ \ con\ \ base
\end{array}
$$

q.e.d. □

8.3. Proof of Theorem 5

NOTATION 7 We consider nested lists with possibly different length of dimensions and corresponding projections:

$$
\begin{array}{rcl}
[\alpha]_{n_1 \times \cdots \times n_d} & \stackrel{\text{def}}{=} & [\ [\alpha]_{n_2 \times \cdots \times n_d}]_{n_1} \\
\pi_{(i_1,\ldots,i_d)} & \stackrel{\text{def}}{=} & \pi_{i_d} \circ \cdots \pi_{i_2} \circ \pi_{i_1} \\
[\alpha]_{(i_1,\ldots,i_n)} & \stackrel{\text{def}}{=} & \pi_{(i_1,\ldots,i_n)}\ [\alpha]
\end{array}
$$

□

8.3.1 Auxiliary Lemmas

NOTATION 8 For a two-dimensional list x, we denote:

$$x^t = chdim^{(1,2)}\ x$$

□

LEMMA 9 *For two-dimensional lists $x, y \in [\alpha]_{n \times m}$:*

$$zip\ (+\!\!+)\,(x^t, y^t) = (x +\!\!+\, y)^t \tag{20}$$

PROOF

$$
\begin{array}{ll}
& (zip\ (+\!\!+)(x^t, y^t))_{(i,j)} \\
= & \quad \{\ \text{Definition of } zip\ \} \\
& ((x^t)_i +\!\!+\, (y^t)_i)_j \\
= & \quad \{\ \text{Property of concatenation}\ \} \\
& \begin{cases} (x^t)_{(i,j)} & \text{if } j \le l \stackrel{\text{def}}{=} |(x^t)_i| \\ (y^t)_{(i,j-l)} & \text{else} \end{cases} \\
= & \quad \{\ \text{Definition of } chdim\ \} \\
& \begin{cases} x_{(j,i)} & \text{if } j \le l = |x| \\ y_{(j-l,i)} & \text{else} \end{cases} \\
= & \quad \{\ \text{Definitions of } chdim \text{ and concatenation}\ \} \\
& ((x +\!\!+\, y)^t)_{(i,j)}
\end{array}
$$

q.e.d. □

LEMMA 10 *For a distributable homomorphism $h = \oplus \updownarrow \otimes$, and $z \in [\alpha]_{n \times m}$:*

$$(zip\,\oplus) \updownarrow (zip\,\otimes)\ z = \big(map\ h\ (z^t)\big)^t \tag{21}$$

PROOF Induction on $|z|$. Base case $|z| = n = 1$:

$$
\begin{aligned}
(zip\,\oplus) \updownarrow (zip\,\otimes)\ [[a_1,\ldots,a_m]] &= [[a_1,\ldots,a_m]] \\
&= [[a_1],\ldots,[a_m]]^t = (map\ h\ [[a_1],\ldots,[a_m]])^t \\
&= \big(map\ h\ [[a_1,\ldots,a_m]]^t\big)^t
\end{aligned} \tag{22}
$$

Induction step: let $|z| > 1$, $z = x +\!\!+\, y$ and $g = (zip\,\oplus) \updownarrow (zip\,\otimes)$

$$
\begin{aligned}
& (g(x + y))_{(i,j)} \\
= & \quad \{ \text{ Definition of DH } \} \\
& \big(\mathit{zip}^2(\oplus)(g\ x, g\ y) + \mathit{zip}^2(\otimes)(g\ x, g\ y)\big)_{(i,j)} \\
= & \quad \{ \text{ Induction hypothesis } \} \\
& \Big(\mathit{zip}^2(\oplus)\,((\mathit{map}\ h\ x^t)^t, (\mathit{map}\ h\ y^t)^t) + \\
& \quad \mathit{zip}^2(\otimes)\,((\mathit{map}\ h\ x^t)^t, (\mathit{map}\ h\ y^t)^t)\Big)_{(i,j)} \\
= & \quad \{ \text{ Definition of } + \text{ and } \mathit{zip},\ l = |(\mathit{map}\ h\ x^t)^t|\ \} \\
& \begin{cases} \big(\mathit{zip}^2(\oplus)\,((\mathit{map}\ h\ x^t)^t, (\mathit{map}\ h\ y^t)^t)\big)_{(i,j)} & \text{if } i \leq l \\ \big(\mathit{zip}^2(\otimes)\,((\mathit{map}\ h\ x^t)^t, (\mathit{map}\ h\ y^t)^t)\big)_{(i-l,j)} & \text{otherwise} \end{cases} \\
= & \quad \{ \text{ Def. of } \mathit{zip} \text{ and } \mathit{chdim},\ l = \big|(\mathit{map}\ h\ x^t)_{(1)}\big|\ \} \\
& \begin{cases} (\mathit{map}\ h\ x^t)_{(j,i)} \oplus (\mathit{map}\ h\ y^t)_{(j,i)} & \text{if } i \leq l \\ (\mathit{map}\ h\ x^t)_{(j,i-l)} \otimes (\mathit{map}\ h\ y^t)_{(j,i-l)} & \text{otherwise} \end{cases} \\
= & \quad \{\ l = \big|h\ (x^t)_{(1)}\big|\ \} \\
& \begin{cases} (h\ (x^t)_j)_i \oplus (h\ (y^t)_j)_i & \text{if } i \leq l \\ (h\ (x^t)_j)_{(i-l)} \otimes (h\ (y^t)_j)_{(i-l)} & \text{otherwise} \end{cases} \\
= & \quad \{ \text{ Definition of DH } \} \\
& (h\,((x^t)_j + (y^t)_j))_i \\
= & \quad \{ \text{ Definition of } \mathit{zip}\ \} \\
& \Big(h\ \ (\mathit{zip}(+)\,(x^t, y^t))_j\Big)_i \\
= & \quad \{\ (20)\ \} \\
& \big(h\ \ ((x + y)^t)_j\big)_i\ =\ (\mathit{map}\ h\ (x + y)^t)_{(j,i)} \\
= & \quad \{ \text{ Definition of } \mathit{chdim}\ \} \\
& \big((\mathit{map}\ h\ (x + y)^t)^t\big)_{(i,j)}
\end{aligned}
$$

q.e.d. □

LEMMA 11 *For a function $f : [\alpha] \to [\alpha]$, and $a \leq d$:*

$$
\mathit{map}\ (\mathit{mapdim}^{(a,d)}\ f)\ =\ \mathit{mapdim}^{(a+1,d+1)}\ f \tag{23}
$$

PROOF

$$
\begin{aligned}
& \mathit{map}\ (\mathit{mapdim}^{(a,d)}\ f) \\
= & \quad \{ \text{ Definition of } \mathit{mapdim}\ \} \\
& \mathit{map}\,\big(\mathit{chdim}^{(a,d)}\ \circ\ \mathit{map}^{d-1}\ f\ \circ\ \mathit{chdim}^{(a,d)}\big) \\
= & \quad \{\ (15)\ \} \\
& \mathit{map}\,\big(\mathit{chdim}^{(a,d)}\big)\ \circ\ \mathit{map}^d\ f\ \circ\ \mathit{map}\,\big(\mathit{chdim}^{(a,d)}\big) \\
= & \quad \{\ \mathit{map}\big(\mathit{chdim}^{(n,m)}\big) = \mathit{chdim}^{(n+1,m+1)}\ \} \\
& \mathit{chdim}^{(a+1,d+1)}\ \circ\ \mathit{map}^d\ f\ \circ\ \mathit{chdim}^{(a+1,d+1)} \\
= & \quad \{ \text{ Definition of } \mathit{mapdim}\ \} \\
& \mathit{mapdim}^{(a+1,d+1)}\ f
\end{aligned}
$$

q.e.d. □

Lemma 12 *For a distributable homomorphism h and all $a \le (d-1)$, $d \ge 2$:*

$$\left(map^{d-1}\ \mathit{flat}\right)\ \circ\ mapdim^{(a,d+1)}\ h\ \ =\ \ mapdim^{(a,d)}\ h\ \circ\ \left(map^{d-1}\ \mathit{flat}\right) \tag{24}$$

Proof

$$
\begin{aligned}
& map^{d-1}\ \mathit{flat}\ \circ\ mapdim^{(a,d+1)}\ h\\
= \quad & \{\ \text{Substituting } chdim^{(a,d-1)}\ \circ\ mapdim^{(d-1,d+1)}\ h\ \circ\ chdim^{(a,d-1)}\ =\ mapdim^{(a,d+1)}\ \}\\
& map^{d-1}\ \mathit{flat}\ \circ\ chdim^{(a,d-1)}\ \circ\ mapdim^{(d-1,d+1)}\ h\ \circ\ chdim^{(a,d-1)}\\
= \quad & \{\ map^{d-1}\ \mathit{flat}\ \circ\ chdim^{(a,d-1)}\ =\ chdim^{(a,d-1)} \circ map^{d-1}\ \mathit{flat}\ \}\\
& chdim^{(a,d-1)} \circ map^{d-1}\ \mathit{flat}\ \circ\ mapdim^{(d-1,d+1)}\ h\ \circ\ chdim^{(a,d-1)}\\
= \quad & \{\ \text{Equality (23)}\ \}\\
& chdim^{(a,d-1)} \circ map^{d-2}\left(map\ \mathit{flat} \circ mapdim^{(1,3)}\ h\right)\ \circ\ chdim^{(a,d-1)}\\
= \quad & \{\ map\ \mathit{flat} \circ mapdim^{(1,3)}\ h\ =\ mapdim^{(1,2)}\ h\ \circ\ map\ \mathit{flat}\ \}\\
& chdim^{(a,d-1)}\ \circ\ map^{d-2}\left(mapdim^{(1,2)}\ h\ \circ\ map\ \mathit{flat}\right)\ \circ\ chdim^{(a,d-1)}\\
= \quad & \{\ \text{Equality (23) backwards}\ \}\\
& chdim^{(a,d-1)}\ \circ\ mapdim^{(d-1,d)}\ h\ \circ\ map^{d-1}\ \mathit{flat}\ \circ\ chdim^{(a,d-1)}\\
= \quad & \{\ map^{d-1}\ \mathit{flat}\ \circ\ chdim^{(a,d-1)}\ =\ chdim^{(a,d-1)}\ \circ\ map^{d-1}\ \mathit{flat}\ \}\\
& chdim^{(a,d-1)}\ \circ\ mapdim^{(d-1,d)}\ h\ \circ\ chdim^{(a,d-1)}\ \circ\ map^{d-1}\ \mathit{flat}\\
= \quad & \{\ \text{Removing auxiliary } chdim^{(a,d-1)}\ \}\\
& mapdim^{(a,d)}\ h\ \circ \left(map^{d-1}\ \mathit{flat}\right)
\end{aligned}
$$

q.e.d. □

8.3.2 Theorem

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

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

$$
\begin{aligned}
& \mathit{flat}\ \circ\ mapdim^{(1,2)}\ h\ \circ\ mapdim^{(2,2)}\ h\ \circ\ nest\\
= \quad & \{\ \text{(21), Definition of } mapdim\ \}\\
& \mathit{flat}\ \circ\ (zip \oplus) \updownarrow (zip \otimes)\ \circ\ map\ h\ \circ\ nest\\
= \quad & \{\ \text{(19), (18)}\ \}\\
& \mathit{flat}\ \circ\ nest\ \circ\ h\ \circ \mathit{flat}\ \circ\ nest\ =\ h
\end{aligned}
$$

Induction step: let $d > 1$:

$$
\begin{aligned}
& \mathit{flat}^d\ \circ\ \bigcirc_{i=1}^{d+1} mapdim^{(i,d+1)}\ h\ \circ\ nest^d\\
= \quad & \{\ \text{Let } A\ =\ \mathit{flat}^{d-1}\ \circ\ \bigcirc_{i=1}^{d-1} mapdim^{(i,d)}\ h \text{ after applying } \mathit{flat}^d = \mathit{flat}^{d-1}\ \circ\ map^{d-1}\ \mathit{flat} \text{ and (24)}\ \}\\
& A\ \circ\ map^{d-1}\ \mathit{flat}\ \circ\ mapdim^{(d,d+1)}\ h\ \circ\ mapdim^{(d+1,d+1)}\ h\ \circ\ nest^d\\
= \quad & \{\ \text{(15), (23)}\ \}\\
& A\ \circ\ map^{d-1}\ \mathit{flat}\ \circ\ map^{d-1}\left(mapdim^{(1,2)}\ h\ \circ\ mapdim^{(2,2)}\ h\right)\ \circ\ nest^d
\end{aligned}
$$

$=\quad$ { Definition of *nest* and (15) }

$A \circ map^{d-1}\ \Big(flat \circ mapdim^{(1,2)}\ h \circ mapdim^{(2,2)}\ h \circ nest\Big) \circ nest^{d-1}$

$=\quad$ { Base case $d=1$ }

$A \circ map^{d-1}\ h \circ nest^{d-1}$

$=\quad$ { Definition of *mapdim* }

$flat^{d-1} \circ \bigcirc_{i=1}^{d} mapdim^{(i,d)}\ h \circ nest^{d-1}$

$=\quad$ { Induction hypothesis }

h

q.e.d. □

8.4. Proof of Theorem 6

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

$$\bigcirc_{i=k}^{d+1}\big(chdim^{(i,d+1)} \circ map^{d}\ h \circ chdim^{(i,d+1)}\big) = \bigcirc_{i=k}^{d} chdim^{(i,i+1)} \circ \bigcirc_{i=k}^{d+1}\big(map^{d}\ h \circ chdim^{(i,d+1)}\big) \tag{25}$$

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

$\bigcirc_{i=k-1}^{d+1} mapdim^{(i,d+1)}\ h$

$=\quad$ { Definition of the $\bigcirc$-notation and of *mapdim* }

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

$=\quad$ { Induction hypothesis }

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

$=\quad$ { Promote $\bigcirc_{i=k}^{d-1} chdim^{(i,i+1)}$ to the left through independent dimensions }

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

$=\quad$ { $chdim^{(k-1,d+1)} \circ chdim^{(d,d+1)} = chdim^{(k-1,d)} \circ chdim^{(k-1,d+1)}$, promote $map^{d}\ h$ to the right }

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

$=\quad$ { Absorbing right }

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

$=\quad$ { $chdim^{(k-1,d+1)} \circ chdim^{(k-1,d)} = chdim^{(d,d+1)} \circ chdim^{(k-1,d+1)}$; absorb left }

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

$=\quad$ { Apply $chdim^{(s,s+1)} \circ chdim^{(k-1,s+1)} = chdim^{(k-1,s)} \circ chdim^{(s,s+1)}$ for $s = k, \dots, d$ }

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

q.e.d.