Programming with BSP Homomorphisms

Joeffrey Legaux[2], Zhenjiang Hu[1], Frédéric Loulergue[2], Kiminori Matsuzaki[3], and Julien Tesson[4]

[1] National Institute of Informatics, Tokyo, Japan
Hu@nii.ac.jp
[2] LIFO, Université d'Orléans, France
{Joeffrey.Legaux,Frederic.Loulergue}@univ-orleans.fr
[3] Kochi University of Technology, Kochi, Japan
matsuzaki.kiminori@kochi-tech.ac.jp
[4] Université Paris Est, LACL, UPEC, France
Julien.Tesson@lacl.fr

Abstract. Algorithmic skeletons in conjunction with list homomorphisms play an important role in formal development of parallel algorithms. We have designed a notion of homomorphism dedicated to bulk synchronous parallelism. In this paper we derive two application using this theory: sparse matrix vector multiplication and the all nearest smaller values problem. We implement a support for BSP homomorphism in the Orléans Skeleton Library and experiment it with these two applications.

Keywords: Algorithmic skeletons, Constructive algorithms, Bulk synchronous parallelism, All nearest smaller values, Sparse linear algebra.

1 Introduction

Parallel programming needs to be as widespread as parallel machines that now range from smartphones to supercomputers. Structured models of parallelism such as algorithmic skeletons [2] or bulk synchronous parallelism [21], ease the writing and reasoning on parallel programs. Algorithmic skeletons are, or can be seen as, higher-order functions that capture usual parallel patterns but that have a semantics identical or close to usual higher-order functions on collections, in particular lists. The most famous ones are the map and reduce skeletons. Bulk synchronous parallelism offers an abstract and simple model of parallelism yet allowing to take realistically into account the communication costs of parallel algorithms. It has been used in many application domains.

The theory of lists [1] is a powerful tool to systematically develop correct functional programs. From a specification, or naive implementation of a program, it allows to derive step-by-step, a more efficient version. Algorithmic skeletons in conjunction with list homomorphisms (or homomorphisms for short) play an important role in formal development of parallel algorithms [3, 7, 14].

We have defined a notion of homomorphism dedicated to bulk synchronous parallelism, and explored its theory [5, 17] in the context of the Coq proof assistant [18]. Our SDPP [19] framework allows to derive step-by-step correct parallel programs in Coq and then to extract functional parallel programs for the

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 446–457, 2013.

OCaml [10] language and the BSML library [11] that can be compiled and run in parallel. If our long term goal is to provide sufficient automation to use the Coq proof assistant to ease the development of efficient parallel programs, our framework still lacks automation and the purely functional programs we can extract cannot compete yet with high-level C++ hand-written code. Therefore on a practical side it would be interesting to have a support for BSP homomorphisms in an efficient library of algorithmic skeletons such as OSL : the C++ Orléans Skeleton Library [8]. The work presented in this paper provides such a support and we illustrate its use through the derivation of non-trivial applications.

The main technical contributions of this paper can be summarised as follows.

- We derive two applications in a systematic way using the theory of BSP homomorphisms: a sparse-matrix vector multiplication and the all nearest smaller values algorithm;
- We implement support for the execution of BSP homomorphisms in the Orléans Skeleton Library;
- We experiment with these applications implemented with OSL on parallel machines.

The organisation of this paper is as follows. We start by reviewing the basic concepts of homomorphism and recall the definition of the BSP homomorphisms and their theory (section 2). We then show how to derive BSP homomorphisms from specifications in section 3. Section 4 is devoted to the Orléans Skeleton Library, in particular support for BSP homomorphisms with the `bh` skeleton. We experiment with the derived applications in section 5. We discuss the related work in Section 6 and conclude the paper in section 7.

2 BSP Homomorphisms

Our notations are basically based on the programming language Haskell [15]. Functional application is denoted by a space and an argument may be written without brackets. Thus $f\ a$ means $f(a)$. Functions are curried, i.e. functions take one argument and return a function or a value, and the function application associates to the left. Thus $f\ a\ b$ means $(f\ a)\ b$. Infix binary operators will often be denoted by $\oplus$, $\otimes$, $\odot$. Functional application binds stronger than any other operators, so $f\ a \oplus b$ means $(f\ a) \oplus b$, but not $f(a \oplus b)$. Lists are finite sequences of values of the same type. A list is either the empty list, a singleton or a concatenation of two lists. We denote $[]$ for the empty list, $[a]$ for a singleton list with element a, and $x +\!\!+ y$ for a concatenation of two lists x and y. The concatenation operator is associative. Lists are destructed by pattern matching.

Definition 1 (Homomorphism). Function h is said to be a *homomorphism*, if it is defined recursively in the form of

$$h\ [\,] = id_{\odot} \qquad h\ [a] = f\ a \qquad h\ (x +\!\!+ y) = h(x) \odot h(y)$$

where $id_{\odot}$ denotes the identity unit of $\odot$. Since h is uniquely determined by f and $\odot$, we will write $h = (\!|\odot, f|\!)$.

Definition 2 (BSP Homomorphism (BH)). Given a function k, and two homomorphisms $g_1 = (\!|\oplus, f_1|\!)$, $g_2 = (\!|\otimes, f_2|\!)$[1], h is said to be a *BH*, if it is defined in the following way.

$$\begin{cases} h\ []\ l\ r = [] \\ h\ [a]\ l\ r = [k\ a\ l\ r] \\ h\ (x +\!\!+ y)\ l\ r = h\ x\ l\ (g_1\ y \oplus r)\ +\!\!+\ h\ y\ (l \otimes g_2\ x)\ r \end{cases}$$

The above h defined with functions k, g_1, g_2, and associative operators $\oplus$ and $\otimes$ is denoted as $h = BH(k, (\!|\oplus, f_1|\!), (\!|\otimes, f_2|\!))$.

Function h is a *higher-order* homomorphism, which computes on a list and returns a new list of the same length. In addition to the input list, h has two additional parameters, l and r, which append necessary information to perform computation on the list. The information of l and r, as defined in the third equation, is propagated from left and right with functions $g_2, \otimes$ and $g_1, \oplus$ respectively. By definition, a BH can be computed in parallel since it is a composition of local computations and of homomorphisms which can be easily parallelised [3].

Rather than checking directly that a function is a *BH* we use an indirect way using the *mapAround* function. *mapAround*, compared to *map*, captures more interesting independent computations on each element of lists. Intuitively, *mapAround* maps a function to each element (of a list) but is allowed to use information of the sublists on the left and right of the element, e.g.,

$$\begin{aligned} &mapAround\ f\ [x_1, x_2, \ldots, x_n] \\ &= [f\ ([], x_1, [x_2, \ldots, x_n]), f\ ([x_1], x_2, [x_3, \ldots, x_n]), \ldots, f\ ([x_1, \ldots, x_{n-1}], x_n, [])]. \end{aligned}$$

Theorem 1 (Parallelization *mapAround* with *BH*). For a function $h = mapAround\ f$, if we can decompose f as $f\ (ls, x, rs) = k\ (g_1\ ls, x, g_2\ rs)$ where g_i is a composition of a function p_i with a homomorphism, $g_i\ x = p_i((\!|\oplus_i, k_i|\!)\ x)$, then

$$h\ xs = BH(k', (\!|\oplus_1, k_1|\!), (\!|\oplus_2, k_2|\!))\ xs\ \iota_{\oplus_1}\ \iota_{\oplus_2}$$

where $k'\ (l, x, r) = k(p_1\ l, x, p_2\ r)$ holds, where $\iota_{\oplus_1}$ is the (left) unit of $\oplus_1$ and $\iota_{\oplus_2}$ is the (right) unit of $\oplus_2$.

Proof. This can be proved by induction on the input list of h. The detailed proof in Coq is discussed in [5,17].

Theorem 1 is general and powerful in the sense that it can parallelize not only *mapAround* but also other collective functions, such as *scan*, to *BH* [5,17].

3 Program Derivation Using BSP Homomorphisms

In this section, we demonstrate with two nontrivial examples how to derive applications using the BH theory. One is the all nearest smaller values problem and the other is the sparse matrix-vector multiplication.

[1] See [17] for a discussion about weaker conditions for the definition of BSP homomorphism.

3.1 All Nearest Smaller Values

The All Nearest Smaller Values (ANSV) problem is as follows:

> Let $as = [a_1, a_2, \ldots, a_n]$ be an array of elements from a totally ordered domain. For each a_j, find the nearest element to the left of a_j and the nearest element to the right of a_j that are less than a_j. If there is no such an element, we put $-\infty$ instead.

An example of the input and the output for the function *ansv* that solves this problem is as follows.

$$\begin{aligned}&ansv\ [3,1,4,1,5,9,2,6,5]\\ &= [(-\infty,1),(-\infty,-\infty),(1,1),(-\infty,-\infty),(1,2),(5,2),(1,-\infty),(2,5),(2,-\infty)]\end{aligned}$$

A direct specification of the ANSV algorithm is as follows:

$$\begin{aligned}&ansv\ as = mapAround\ nsv\ as\\ &\qquad \textbf{where}\ nsv\ (ls,x,rs) = (nsvL\ x\ ls, nsvR\ x\ rs)\\ &\qquad\qquad nsvL\ x\ [\,] = -\infty\\ &\qquad\qquad nsvL\ x\ (ls +\!\!+ [l]) = \textbf{if}\ l < x\ \textbf{then}\ l\ \textbf{else}\ nsvL\ x\ ls\\ &\qquad\qquad nsvR\ x\ [\,] = -\infty\\ &\qquad\qquad nsvR\ x\ ([r] +\!\!+ rs) = \textbf{if}\ r < x\ \textbf{then}\ r\ \textbf{else}\ nsvR\ x\ rs\end{aligned}$$

where we simply use *mapAround* to compute on each element and its surround (left and right arrays) with the function *nsv*. In the definition of *nsv*, *nsvL x ls* is to compute the rightmost element in *ls* that is less than x, while *nsvR x rs* computes the leftmost element in *rs* that is less than x.

However, to use the Theorem 1, the computations on the left and right arrays need to be expressed as homomorphisms independent from the center element (this precondition derives from the function shape requirement in the theorem). We can give such a definition where we first select the candidates from the left and right arrays and then choose a correct one from them. Since the computation for the left and right is symmetrical, we here discuss the right one.

We are looking for the nearest smaller values, thus we can discard values that are equal or superior to the previous elements and only retain a sample of decreasing candidates as shown in Figure 1. Therefore, we can decompose the definition of *nsvR* as follows into a homomorphism that removes unnecessary elements from an array and a function that picks up the nearest smaller value. Since the result of the homomorphism is a list in which elements are in decreasing order, the binary operator of the homomorphism just removes elements from the right list that are larger than the rightmost element.

$$\begin{aligned}&nsvR\ v\ rs = pickupR\ v\ ((\!|\oplus, id|\!)\ rs)\\ &\qquad \textbf{where}\ (ls +\!\!+ [l]) \oplus rs = ls +\!\!+ [l] +\!\!+ dropWhile\ (\lambda x.x > l)\ rs\\ &\qquad\qquad pickupR\ x\ [\,] = -\infty\\ &\qquad\qquad pickupR\ x\ ([r] +\!\!+ rs) = \textbf{if}\ r < x\ \textbf{then}\ r\ \textbf{else}\ pickupR\ x\ rs\end{aligned}$$

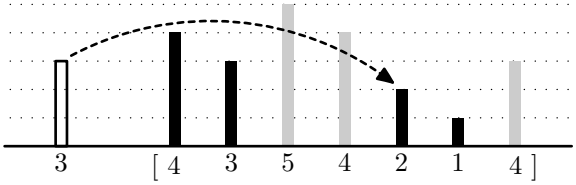

Fig. 1. The candidates in a right array. The values that keep a decreasing order are kept as candidates (in black), while the others are discarded (in gray). Here the value 2 will be kept as the final solution since it is the closest candidate that is inferior to the center value 3 (in white).

We have decomposed *nsvR* into a function *pickupR* and a homomorphism $(\!|\oplus, id|\!)$; the function *nsvL* can be similarly decomposed into *pickupL* and $(\!|\otimes, id|\!)$. Thus we can rewrite *nsv* as follows :

$$\begin{aligned} nsv\ (ls, x, rs) &= k\ ((\!|\otimes, id|\!)\ ls, x, (\!|\oplus, id|\!)\ rs) \\ &\quad \textbf{where}\ k\ (l, x, r)\ =\ (pickupL\ x\ l,\ pickupR\ x\ r) \end{aligned}$$

This form matches the one needed to apply the Theorem 1 in order to derive the ANSV into a *BH*.

3.2 Sparse Matrix-Vector Multiplication

Sparse matrices are often compressed into array representations. We develop a parallel program to compute the multiplication of a sparse matrix and a vector.

Here we consider an array representation that consists of triples (y, x, a):

- y: the row-index of the nonzero element,
- x: the column-index of the nonzero element, and
- a: the value of the nonzero element.

We assume that elements are sorted with respect to the indices y and x. For example, the following matrix A is represented by the array *as* with five triples.

$$A = \begin{pmatrix} 1.1 & 2.2 & 0 \\ 0 & 1.3 & 1.4 \\ 0 & 0 & 3.5 \end{pmatrix} \qquad \begin{aligned} as = [&(0,0,1.1),\ (0,1,2.2),\\ &(1,1,1.3),\ (1,2,1.4),\ (2,2,3.5)] \end{aligned}$$

In the matrix-vector multiplication, there is a result element for each row. Let us put the result on the first element in the row, and clear the other values with a dummy value denoted as $\square$. For example, multiplying a vector $[3.0, 4.0, 1.0]$ to the array representation *as* yields

$$mult\ as\ [3.0, 4.0, 1.0] = [(0,0,12.1),\ (0,1,\square),\ (1,1,6.6),\ (1,2,\square),\ (2,2,3.5)]\ .$$

Note that we can apply the array packing [5] to compact the result into the result vector $[12.1, 6.6, 3.5]$.

Now we develop the specification of this problem using the *mapAround* function. The first and important step is to determine which kind of values are needed from the left or from the right. To check whether an element is the first one in the row, we simply compare the row-index of the element with that of the left element. When we compute the result value, we need the partial sum of the rightward values in the row, multiplied by the vector. Therefore, the values passed from the right are the row-index of the right element and the partial sum in the row (of right element). Based on these insights, we can develop a specification with the *mapAround* function. In the following program, $v\langle i\rangle$ denotes the i^{th} element of the vector v.

$$
\begin{array}{l}
\mathit{mult}\ \mathit{as}\ v = \mathit{mapAround}\ (f\ v)\ \mathit{as} \\
\textbf{where}\ f\ v\ (ls, (y, x, a), rs) = \textbf{let}\ y_l = g_l\ ls; (y_r, s_r) = g_r\ v\ rs \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in if}\ (y_l == y)\ \textbf{then}\ (y, x, \Box) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{elseif}\ (y_r == y)\ \textbf{then}\ (y, x, v\langle x\rangle * a + s_r) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ (y, x, v\langle x\rangle * a)
\end{array}
$$

Now we give the definition of the auxiliary functions and check that they are homomorphisms. The function g_l just takes the row-index of the last element in a list. It is a homomorphism

$$g_l = (\!|\gg, \lambda(x, y, a).y|\!) \quad \textbf{where}\ \ a \gg b = b\ ,$$

and any value (here we use -1) is a left unit of the operator $\gg$. The function $g_r\ v$ is a bit more complicated and is defined as follows.

$$
\begin{array}{l}
g_r\ v\ [(y, x, a)] = (y, a * v\langle x\rangle) \\
g_r\ v\ [as +\!\!+ (y, x, a)] = \textbf{let}\ (y', s) = g_r\ v\ as \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (y', \textbf{if}\ y' == y\ \textbf{then}\ s + a * v[x]\ \mathit{else}\ s)
\end{array}
$$

This function is indeed a homomorphism as follows.

$$
\begin{array}{l}
g_r\ v\ [(y, x, a)] = (y, a * v\langle x\rangle) \\
g_r\ v\ (ls +\!\!+ rs) = g_r\ v\ ls \odot g_r\ v\ rs \\
\qquad \textbf{where}\ (y_l, s_l) \odot (y_r, s_r) = \textbf{if}\ y_l == y_r\ \textbf{then}\ (y_l, s_l + s_r)\ \textbf{else}\ (y_l, s_l)
\end{array}
$$

A right unit of the operator $\odot$ is $(-1, 0)$.

Now we can apply the Theorem 1 to the specification above and obtain the following BH.

$$
\begin{array}{l}
\mathit{mult}\ \mathit{as}\ v = BH(k\ v, (\!|\odot, \lambda(y, x, a).(y, a * v\langle x\rangle)|\!), (\!|\gg, \lambda(x, y, a).y|\!))\ \mathit{as} \\
\quad \textbf{where}\ k\ v\ (y_l, (y, x, a), (y_r, s)) = \textbf{if}\ y == y_l\ \textbf{then}\ (y, x, \Box) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{elseif}\ y == y_r\ \textbf{then}\ (y, x, a * v\langle x\rangle + s) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ (y, x, a * v\langle x\rangle) \\
\qquad a \gg b = b \\
\qquad (y_l, s_l) \odot (y_r, s_r) = \textbf{if}\ y_l == y_r\ \textbf{then}\ (y_l, s_l + s_r)\ \textbf{else}\ (y_l, s_l)
\end{array}
$$

4 BH in the Orléans Skeleton Library

4.1 An Overview of Orléans Skeleton Library

Orléans Skeleton Library is a C++ library of data-parallel algorithmic skeletons. It is implemented on top of MPI and takes advantage of the expression templates optimisation techniques [22] to be very efficient yet allowing programming in a functional style. Programming with OSL is very similar to programming in sequential as OSL offers a global view of parallel programs [4]. OSL programs operate on *distributed arrays* that are one dimensional arrays such that, at the time of the creation of the array, data is distributed among the processors. Distributed arrays are implemented as a template class `DArray<T>`. A distributed array consists of `bsp_p` partitions, where `bsp_p` is the number of processing elements of the parallel (BSP) machine. Each partition is an array of elements of type `T`.

To give a quick, yet precise, overview of OSL, Fig. 2 presents an informal semantics for the main OSL skeletons together with their signatures. In this figure, `bsp_p` is noted p. A distributed array of type `DArray<T>` can be seen "sequentially" as an array $[t_0, \ldots, t_{t.size-1}]$ where $t.size$ is the (global) size of the (distributed) array t (and we use the same notation if t is a C++ vector). But as with the `getPartition` skeleton, the user can expose the distribution of the distributed array, this informal semantics should also indicates how the array is distributed. We write the distribution as a subscript D of the distributed array. D is a function from $\{0, \ldots, \texttt{bsp\_p} - 1\}$ to $\mathbb{N}$.

The first skeleton, `map` (and variants such as `zip`, `mapIndex`, *etc.*) is the usual combinator used to apply a function to each element of a distributed array (or two for `zip`). The first argument of both `map` and `zip` could be a C++ functor either extending `std::unary_function` or `std::binary_function`, respectively.

Parallel reduction and parallel prefix computation with a binary *associative* operator $\oplus$ are performed using respectively the `reduce` and `scan` skeletons. Communications are needed to execute both skeletons.

`permute` and `shift` are communication skeletons. The next skeleton only modifies the distribution of the distributed array, not its content: `redistribute` distributes the content of the distributed array according to a vector of integers representing the target distribution. All the skeletons up to `redistribute` preserve the distribution. It means that if they are applied to evenly distributed arrays, the result will be an evenly distributed array. The `redistribute` skeleton may thus seems useless. However, some algorithms such as BSP regular sampling sort, require intermediate and final results that are not evenly distributed. To implement such algorithms, two additional skeletons are needed: `getPartition` and `flatten`. The `getPartition` skeleton exposes how a distributed array is distributed among the processors, while `flatten` is the inverse operation.

As a very short OSL example program, we can compute the variance $\sum_{i=0}^{n-1}(x_i - \frac{\sum_{j=0}^{n-1} x_j}{n})$ of a sequence of random variables x_i:

Skeleton	Signature Informal semantics
`map`	`DArray<W> map(W f(T), DArray<T> t)` $\texttt{map}(f, [t_0, \ldots, t_{t.size-1}]_D) = [f(t_0), \ldots, f(t_{t.size-1})]_D$
`reduce`	`<T> reduce(T⊕(T,T), DArray<T> t)` $\texttt{reduce}(\oplus, [t_0, \ldots, t_{t.size-1}]_D) = t_0 \oplus t_1 \oplus \ldots \oplus t_{t.size-1}$
`scan`	`DArray<T> scan(T⊕(T,T), DArray<T> t)` $\texttt{scan}(\oplus, [t_0, \ldots, t_{t.size-1}]_D) = [\oplus_{i=0}^{0} t_i; \ldots; \oplus_{i=0}^{t.size-1} t_i]_D$
`permute`	`DArray<T> permute(int f(int), DArray<T> t)` $\texttt{permute}(f, [t_0, \ldots, t_{t.size-1}]_D) = [t_{f^{-1}(0)}, \ldots, t_{f^{-1}(t.size-1)}]_D$
`shift`	`DArray<T> shift(int o, T f(T), DArray<T> t)` $\texttt{shift}(o, f, [t_0, \ldots, t_{t.size-1}]_D) = [f(0), \ldots, f(o-1), t_0, \ldots, t_{t.size-1-o}]_D$
`redistribute`	`DArray<T> redistribute(Vector<int> dist, DArray<T> t)` $\texttt{redistribute}(\texttt{dist}, [t_0, \ldots, t_{t.size-1}]_D) = [t_0, \ldots, t_{t.size-1}]_{\texttt{dist}}$
`getPartition`	`DArray< Vector<T> > getPartition(DArray<T> t)` $\texttt{getPartition}([t_0, \ldots, t_{t.size-1}]_D)$ $= \left[[t_0, \ldots, t_{D(0)-1}], \ldots, [t_{j_i}, \ldots, t_{j_i+D(i)-1}], \ldots, [t_{j_{p-1}}, \ldots, t_{t.size-1}]\right]_{E_p}$ where $E_p(i) = 1$ and $j_i = \sum_{k=0}^{k=i-1} D(k)$
`flatten`	`DArray<T> flatten(DArray< Vector<T> > t)` $\texttt{flatten}([a_0, \ldots, a_{a.size-1}]_D)$ $= \left[a_0[0], \ldots, a_0[a_0.size-1], a_1[0], \ldots, a_{a.size-1}[a_{a.size-1}.size-1]\right]_{D'}$ where $D'(i) = \sum_{j_i \le k < j_i + D(i)} a_k.size$ and $j_i = \sum_{k=0}^{k=i-1} D(k)$

Fig. 2. OSL Skeletons

```
double avg = osl::reduce(std::plus<double>(), x) / x.getGlobalSize();
double variance = osl::reduce(std::plus<double>(),
                  osl::map(boost::bind(std::minus<double>(),avg, _2), x));
```

4.2 Using the BH Skeleton

The signature of the `bh` skeleton is:

```
DArray<typename K::result_type>
  bh(K k, Homomorphism<T, L> * hl, Homomorphism<T, R> * hr,
     L l, R r, const DArray<T>& temp)
```

According to Definition 2, a BH is defined by a function `k` and two homomorphisms `g1` and `g2`, which are applied on a list (in the form of a distributed array `temp`) with two boundary elements `L` and `R`.

`k` can be easily implemented as a usual functor whose `()` operator takes three arguments: the left summary (which will be the result of the application of `g1` on the left part of the list), the current element and the right summary. For `g1` and `g2`, we define a generic virtual base class `Homomorphism` which defines the needed function f, operator $\odot$ and its unit $id_\odot$ (Definition 1). The user can then implement its own homomorphism by creating a derived class that provides concrete implementations of those 3 items.

`k`, `g1` and `g2` are the first three parameters of our generic BH skeleton. To apply it to actual data, we need to provide three last arguments: the boundary

elements `L` and `R`, and the list in the form of a `DArray`. The return value will be a list of the same size, whose type of elements will be the result type of `k`.

Implementing for example the computation of the prefix-sum on an array of integers can be easily done. First, we need the left homomorphism that subsequently adds all the values:

```
class HAdd: public Homomorphism<int, int> {
    public:
        HAdd() { neutral = 0;}
        inline int f(const int& i) {return i;}
        inline int o(const int& i1, const int& i2) {return i1+i2;}
};
```

We do not have any computation to conduct on the right side. However we still need to provide an homomorphism to the `bh` skeleton, so we can define one that always returns the same value. This homomorphism, named `HConst`, is defined in a similar way than `HAdd` but with each operator returning `0`.

We now only have to define the `k` function which will simply add the computed sum of the left sub-array with the current element:

```
struct AddLeft {
  typedef int result_type;
  inline int operator()(int l, int i, int r) const { return l+i; }
};
```

We can now apply the skeleton to compute the prefix sum on any distributed array `d`, using zeros for the boundary values:

```
DArray<int> result = osl::bh(AddLeft(),new HAdd(), new HConst(), 0, 0, d);
```

4.3 Implementation of the BH Skeleton

The `bh` skeleton is implemented with the usual expression template mechanism of our library, so it can be integrated seamlessly in any OSL expression and trigger the fusion optimisation when it is relevant. The recursive definition of homomorphisms provides room for a major optimisation. If we apply the definition to an array of elements, we can write the third recursive rule as such:

$$h\ [x_1, \ldots, x_n] = h\ [x_1, \ldots, x_n - 1] \odot h\ [x_n] = h\ [x_1, \ldots, x_n - 1] \odot f\ (x_n).$$

This allows us to pre-compute locally the application of the homomorphism to each sub-array in a linear time as we only have to apply f and $\odot$ once per element. Without this optimisation, we would have to conduct these operations i times for each of the n x_i elements, thus resulting in a square complexity. Thanks to the associativity of homomorphisms, we can symmetrically implement the same optimisation for the right homomorphism that applies on the end of the array.

A disadvantage is that in order to achieve this purpose we have to consider the local part of the array on each node in its entirety: This forces us to break the loop fusion mechanism, which is based on the fact that each element of the array is treated separately. However fusion can still occur on the expression (if there actually exists one) that produces the input array `temp`.

5 Experiments

We implemented programs computing the ANSV and sparse-matrix vector multiplication using our implementation of the BH skeleton in the OSL library. We then measured the scalabity of those programs when parallelised over several cores on two architectures : a shared-memory computer containing 4 processors with 12 computer cores each (thus a total of 48 cores), and a distributed-memory cluster of 8 machines each containing 2 processors of 4 cores (for a total of 64 cores). More experiments are currently undergoing on a larger cluster containing several hundreds cores. Those measures were conducted using a statistical evaluation protocol [20] in order to ensure stability and reproducibility of the results. ANSV was solved on a 10^7 elements array. Sparse matrix-vector multiplication was conducted on a 10^9 elements matrix with 10% of non-zero elements, leading to an actual 10^8 elements of data.

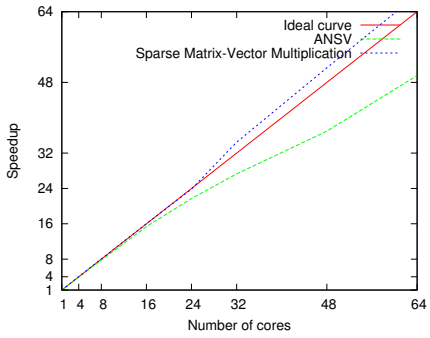

Fig. 3. Distributed memory

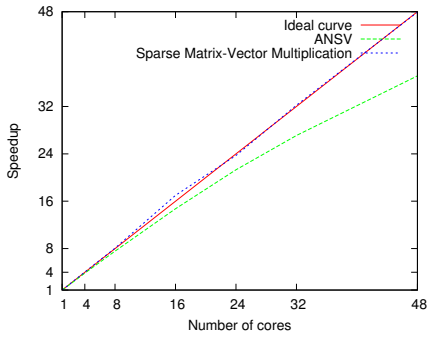

Fig. 4. Shared memory

The ANSV problem scales well although sub-linearly, we may expect its performance to peak at a greater number of cores. This could be explained by the fact that each processor has to communicate its local array of candidate elements to every other core. Those arrays can reach a consequent size on big problems, and the cost of this communication operation may rapidly overcome the parallelisation gains on larger numbers of cores.

On the other hand, the sparse matrix-vector multiplication is perfectly linear. As in this problem the processors only have to exchange a pair of numbers, the communication cost is probably too small to impact the scaling of the algorithm at this level. We also get super-linear speedups on the distributed architecture with a large number of cores, which seems to indicate that this particular computation is limited by the memory bandwidth on the shared memory architecture.

6 Related Work

There are many algorithmic skeletons libraries, for various host languages: [6] is a recent survey of such libraries. Depending on the supported data structures,

these libraries could be used to implement programs obtained by systematic developement based on the theory of lists [3, 14], trees [13] or arrays. However none support BSP homomorphisms. Compared to BSP implementations of skeletons [23] together with usual theories, our theoretical framework and OSL library allow to derive and implement efficient programs such as the all nearest smaller values program.

Several researchers worked on formal semantics for BSP computations, for example [9,16]. But to our knowledge none of these semantics was used to generate programs as the last step of a systematic development. LOGS is a semantics of BSP programs and was used to generate C programs [24]. The main difference with our approach is that it starts from a local and imperative view of parts of the program to build a larger one, and we start from a global and functional view and refine it.

7 Conclusion and Future Work

The theory of bulk synchronous parallel homomorphism allows to derive non-trivial applications. The support of BSP homomorphism in the Orléans Skeleton Library through the *BH* skeleton can be used to implement such applications. In the SkeTo and OSL libraries, fusion [12] is done by the expression templates technique. More global optimisations could be done, in particular using the Proto framework for C++: This is planned. However we still need to investigate the theory of fusion for BSP homomorphisms before incorporing *BH* fusion in OSL.

Acknowledgements. This work is partly supported by ANR (France) and JST (Japan) (project PaPDAS ANR-2010-INTB-0205-02 and JST 10102704). Joeffrey Legaux is supported by a PhD grant from the *Conseil Général du Loiret.*

References

1. Bird, R.: An introduction to the theory of lists. In: Broy, M. (ed.) Logic of Programming and Calculi of Discrete Design, pp. 5–42. Springer (1987)
2. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press (1989), `http://homepages.inf.ed.ac.uk/mic/Pubs`
3. Cole, M.: Parallel Programming with List Homomorphisms. Parallel Processing Letters 5(2), 191–203 (1995)
4. Deitz, S.J., Callahan, D., Chamberlain, B.L., Snyder, L.: Global-view abstractions for user-defined reductions and scans. In: PPoPP, pp. 40–47. ACM, New York (2006)
5. Gesbert, L., Hu, Z., Loulergue, F., Matsuzaki, K., Tesson, J.: Systematic Development of Correct Bulk Synchronous Parallel Programs. In: International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp. 334–340. IEEE (2010)
6. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Software, Practrice & Experience 40(12), 1135–1160 (2010)

7. Gorlatch, S., Bischof, H.: Formal Derivation of Divide-and-Conquer Programs: A Case Study in the Multidimensional FFT's. In: Mery, D. (ed.) Formal Methods for Parallel Programming: Theory and Applications, pp. 80–94 (1997)
8. Javed, N., Loulergue, F.: Parallel Programming and Performance Predictability with Orléans Skeleton Library. In: International Conference on High Performance Computing and Simulation (HPCS), pp. 257–263. IEEE (2011)
9. Jifeng, H., Miller, Q., Chen, L.: Algebraic laws for BSP programming. In: Bougé, L., Fraigniaud, P., Mignotte, A., Robert, Y. (eds.) Euro-Par 1996, Part II. LNCS, vol. 1124, pp. 359–368. Springer, Heidelberg (1996)
10. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml System release 4.00.0 (2012), `http://caml.inria.fr`
11. Loulergue, F.: Parallel Juxtaposition for Bulk Synchronous Parallel ML. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 781–788. Springer, Heidelberg (2003)
12. Matsuzaki, K., Emoto, K.: Implementing Fusion-Equipped Parallel Skeletons by Expression Templates. In: Morazán, M.T., Scholz, S.-B. (eds.) IFL 2009. LNCS, vol. 6041, pp. 72–89. Springer, Heidelberg (2010)
13. Matsuzaki, K., Hu, Z., Takeichi, M.: Parallelization with tree skeletons. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 789–798. Springer, Heidelberg (2003)
14. Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: Automatic Inversion Generates Divide-and-Conquer Parallel Programs. In: Conference on Programming Language Design and Implementation (PLDI), pp. 146–155. ACM Press (2007)
15. O'Sullivan, B., Stewart, D., Goerzen, J.: Real World Haskell. O'Reilly (2008)
16. Stewart, A., Clint, M., Gabarró, J.: Barrier synchronisation: Axiomatisation and relaxation. Formal Aspects of Computing 16(1), 36–50 (2004)
17. Tesson, J.: Environnement pour le développement et la preuve de correction systématiques de programmes parallèles fonctionnels. Ph.D. thesis, LIFO, University of Orléans (November 2011), `http://hal.archives-ouvertes.fr/tel-00660554/en/`
18. The Coq Development Team: The Coq Proof Assistant, `http://coq.inria.fr`
19. The SDPP Development Team: Systematic Development of Parallel Programs, `http://traclifo.univ-orleans.fr/SDPP`
20. Touati, S.A.A., Worms, J., Briais, S.: The Speedup Test. Tech. Rep. inria-00443839, INRIA Saclay - Ile de France (2010), `http://hal.inria.fr/inria-00443839`
21. Valiant, L.G.: A bridging model for parallel computation. Comm. of the ACM 33(8), 103 (1990)
22. Veldhuizen, T.: Techniques for Scientific C++. Computer science technical report 542, Indiana University (2000)
23. Zavanella, A.: The skel-BSP global optimizer: Enhancing performance portability in parallel programming. In: Bode, A., Ludwig, T., Karl, W.C., Wismüller, R. (eds.) Euro-Par 2000. LNCS, vol. 1900, pp. 658–667. Springer, Heidelberg (2000)
24. Zhou, J., Chen, Y.: Generating C code from LOGS specifications. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 195–210. Springer, Heidelberg (2005)