Distributed Elementwise Processing in D-Clean*

Viktória Zsók[1], Zoltán Horváth[1], Zoltán Hernyák[2]

[1] Eötvös Loránd University, Faculty of Informatics
Department of Programming Languages and Compilers
H-1117 Budapest, Pázmány Péter sétány 1/C, Hungary
zsv@inf.elte.hu, hz@inf.elte.hu
[2] Eszterházy Károly College
Department of Information Technology
H-3300 Eger, Eszterházy tér 1.
aroan@ektf.hu

Abstract

This paper presents a new distributed functional skeleton, the elementwise processing implemented in the D-Clean coordination language.
The skeleton coordinating the distributed elementwise processing is a generalisation of the classical `map` function. It contains three main phases. The first one decomposes the input data according to a very well defined strategy, the second phase applies the elementwise processable function on the divided input, and the final phase recombines the result from the already computed subresults. The step from one phase to another also means a changing on the dimension of the intermediate results. The primitives supplied by the D-Clean language are proved to be appropriate for implementing the elementwise processing phases by parameterising them with suitable elementwise processable function parameters. We provide measurements and conclusions regarding the performance of this algorithm in our distributed environment.

1 INTRODUCTION

Programming theorems are general, parameterised solutions for a wide class of problems [6, 7, 4]. A skeleton is a parameterised algorithmic scheme. Functional skeletons correspond to the generalised specification part of a programming theorem. Skeletons are higher order functions [8, 5, 3] with characteristics like increasing the portability and the abstraction level of the programs as well as generalising the computation patterns. The distributed skeleton described here is the generalisation of the `map` function and it is parameterised by a parallel elementwise processable function.

The D-Clean language and the the D-Box intermediate level language, and the informal semantics were introduced in [1]. D-Clean contains high-level language elements for the coordination of the component functions in a distributed environment. Using these primitives distributed applications can be written hiding the details of the environment.

*Supported by OTKA T037742 and Bolyai Scholarship of the Hungarian Academy of Sciences

The definition of a higher order skeleton is given as D-Clean code, which is a Clean-like language with distributed coordination language components. A skeleton is an abstract definition of the distributed computation and it is parameterized by functions, by type and by data. The computation patterns are identified and described by compositions of coordination elements. The coordination constructs have the role of manipulating and controlling the components written in Clean, which are expressing the pure computational aspects. These computational nodes have well identified subtasks of the original problem.

We use the methodology and the abstract specifications presented in [9, 7] for presenting the algorithm introduced in [4]. The first part presents the theoretic background of the distributed elementwise processing method. The second part describes the implementation details, the measurements and finally the conclusions.

2 ELEMENTWISE PROCESSING - BASIC CONCEPTS

The well known higher order function `map` has as its first argument a function from type `a` to type `b` and applies it elementwise to the elements of a data structure, the elements having type `a`. The generalisation of the concept of computing the result of a function elementwise is given for a class of functions defined over multiple linear data structures. Some special cases for generalization of `map` for multiple arguments are also known in Lisp, Scheme and in other functional languages too (like the `zip2`, `mapcar` functions). Our generalisation is illustrated here by a short example.

Suppose there are two arguments and two results, for example the calculation of the intersection and the union of two sets: $A = \{2,3,6,7\}$ and $B = \{1,3,5,7\}$. If we split the sets into two parts in the following way $A_1 = \{2,3\}, B_1 = \{1,3\}$ and $A_2 = \{6,7\}, B_2 = \{5,7\}$, then the union of A and B can be determined as the disjoint union of the subsets belonging to the same part, i.e.: $A \cup B = (A_1 \cup B_1) \tilde{\cup} (A_2 \cup B_2)$, where $\tilde{\cup}$ denotes the disjoint union operator. Similarly, the intersection will be the disjoint union of the intersection of the subsets of the same part: $A \cap B = (A_1 \cap B_1) \tilde{\cup} (A_2 \cap B_2)$. A correct result is obtained, iff the arguments are split into parts according to the concept of totally disjoint decomposition, i.e. if A_1 and B_1 are disjoint both from A_2 and B_2. We continue to split the parts into smaller parts until we obtain sets consisting of single elements or empty sets. Finally the results can be computed for these slices containing at most one single element, i.e. can be determined ”elementwise”: $A \cap B = (\emptyset \cap \{1\}) \tilde{\cup} (\{2\} \cap \emptyset) \tilde{\cup} (\{3\} \cap \{3\}) \tilde{\cup} (\emptyset \cap \{5\}) \tilde{\cup} (\{6\} \cap \emptyset) \tilde{\cup} (\{7\} \cap \{7\})$ and similarly for the union.

We have chosen value 3 as the first cut-value in our example, i.e. elements less or equal to 3 belong to the first part, elements bigger than 3 belong to the second part. The computation of the result can be done in parallel for these two parts. In general it is not easy to calculate a cut-value which is optimal in respect of a balanced workload [4].

3 FORMAL DEFINITIONS

Let H be an arbitrary set and let $\mathcal{P}(H)$ denote the powerset of H. Denoting by n the number of arguments and by m the number of results, we shall use the following notation:

$$\begin{aligned} X ::= &\quad X_1 \times ... \times X_n, X_i \subseteq \mathcal{P}(H)(i = 1,...,n), \\ Y ::= &\quad Y_1 \times ... \times Y_m, Y_j \subseteq \mathcal{P}(H)(j = 1,...,m), \\ &\quad x, \overline{x}, \overline{\overline{x}} \in X. \end{aligned}$$

Definition 31 Completely disjoint decomposition *$\overline{x}, \overline{\overline{x}}$ are called a completely disjoint decomposition of $x \in X$, if $\forall i \in [1,n] : x_i = \overline{x}_i \cup \overline{\overline{x}}_i$ and $\forall i, j \in [1,n] : \overline{x}_i \cap \overline{\overline{x}}_j = \emptyset$.*

Definition 32 Elementwise processable function *Let $f : X \longmapsto Y$. If for every $x \in X$ for every $\overline{x}, \overline{\overline{x}}$ completely disjoint decomposition of x*

$$f(\overline{x}) \cup f(\overline{\overline{x}}) = f(x), \tag{1}$$
$$f(\overline{x}) \cap f(\overline{\overline{x}}) = \emptyset. \tag{2}$$

holds, then f is called an elementwise processable function.

Let us calculate the value of $y = f(x)$, where f is elementwise processable.

Let us define the slice function sl in the following way: $sl : \mathcal{P}(\{1,\dots,n\}) \times H \longmapsto X$,

$$sl(\{i_1,\dots,i_k\},e)_i := \begin{cases} \{e\}, & \text{if } i \in \{i_1,\dots,i_k\} \\ \emptyset, & \text{if } i \notin \{i_1,\dots,i_k\} \end{cases},$$

where $i_1,\dots,i_n$ is a permutation of the numbers $1,\dots,n$.

Let us calculate $y = f(x)$, where f is elementwise processable. Elementwise processable functions are easy to compute for slices containing only one single element. We are allowed to decompose the arguments into slices and compute partial results for each slice separately. The overall result can be obtained as the disjoint union of these partial results according to the definition.

Let us observe that the time-complexity of the computation of the value of an elementwise processable function f is proportional to the number of elements in the set $U ::= \bigcup_{i \in [1..n]} x_i$. The size of x is defined to be the cardinality of the set U.

4 THE ALGORITHM

Suppose that the sets x_i are represented as strictly monotonous non-empty sequences. The first smallest element of x_i is denoted by $x_i(1)$. The length of x_i is denoted by $x_i.dom$. The subsequence of x_i consists of elements having an index $k : k \in (i, j]$ and denoted by $x_i[i, j]$. In this case by decomposing x into n pairwise completely disjoint blocks and by the help of n processors the overall computation time is reduced. The value of f can be computed independently and parallel for each block and in the end the overall result is obtained as the union of the disjoint sets according to the definition.

The description of the size of the input is done by two numbers:

$$N = \sum_{i=1}^{n} dom(x_i),$$
$$M = |\bigcup_{i=1}^{n} \{x_i(j) | j \in range(x_i)\}|,$$

where N is called the 'bag-size' and M the 'set-size' of the input. We take the computation of f for slices as the dominant cost.

A balanced pairwise completely disjoint decomposition would split x into n parts, where the difference of the set-size of any two parts is at most one. It is easy to show that an elementwise processing is necessary to produce such a balanced decomposition. To resolve this conflict we use unbalanced decompositions further on. On the other hand we apply an algorithm which guarantees that the size of the parts are between two bounds and the distribution of the parts among the processors is dynamic.

At the beginning every processor is given a block, which can be processed sequentially. When it is ready, it is provided with another block until we run out of parts. The algorithm takes no more than $M/k + B$ processing steps with k processors, where B is the 'bag-size' of the largest block. It is worth choosing a small B and if we choose it very small, the number of parts will increase and other costs like cutting cost, communication and administration costs will increase with the number of the parts. The number of parts depends on the size of the smallest part, too, so we should ensure that the difference between the size of the smallest and the largest part is as small as possible. An algorithm was presented in [4], which demonstrates that $b \geq B/8$ (b is the size of the smallest block). The implementation described in Section 4. deals with a special case, with two arguments two values elementwise processing, and applies a simplified version of this general solution.

The formal specification of the problem, the decomposition into subproblems, refinement steps, proofs, details of the solution, cost estimation depending on: the size of the input, on the number of processors, on number arguments and results and on the size of the largest part are described in [4] in details.

5 THE DISTRIBUTED ELEMENTWISE PROCESSING IN D-CLEAN

In the following we present the distributed elementwise processing with two arguments and two values, implemented in the D-Clean coordination language [1].

There are three main phases in the elementwise processing. In the first phase the input data is divided into blocks, orthogonally to the sequences.

At the beginning a function defines the pattern p according which the input streams are split into blocks. Blocks are split into two blocks until the size of every block is less than a constant bl. The cut-values stored in pattern p are calculated by the recursive pattern and delmult functions. The delmult function deletes the unnecessary multiple elements from the pattern list. The total disjoint decomposition for two arguments can be done by splitting the blocks at the middle of its first component. A more complex formula is valid for more than two arguments, see [4]. The concat, sub, zip2s, size, select functions are taken from the Edison-library.

```
divide ::  Int ([Int], [Int]) -> [([Int], [Int])]
divide  bl (x, y)
= zip2s xp yp
   where (xp, yp) = (split p x, split p y)
         p = delmult (pattern bl x)

pattern :: Int [Int] -> [Int]
pattern bl x
   | l <= bl = empty
   | l > bl = concat (concat (concat first f1)
   (concat middle f2)) last
   where
      first = sub x (0, 1)
      middle = sub x (half, halfp)
      last = sub x (l-1, l)
      f1 = pattern bl (sub x (0, halfp))
      f2 = pattern bl (sub x (half, l))
      l = size x
      half = l / 2
      halfp = half + 1
```

The blocks are determined in a way that they form a total disjoint decomposition of the input. Therefore all the blocks can be processed in completely independent way of each other in the second phase of the computation. The ewp2a2v function called by the solve function is used to process a block. ewp2a2v makes slices of the block using the slice function, applies the elementwise processable function f on the slices and collects the partial results.

```
slice :: [Int] [Int] -> (([Int], [Int]), [Int], [Int])
slice x y
  | nx == 0 = (( empty, sy), empty, ym)
  | ny == 0 = (( sx, empty), xm, empty)
  | xe == ye = (( sx, sy), xm, ym)
  | xe < ye = (( sx, empty), xm, y)
  | xe > ye = (( empty, sy), x, ym)
  where
```

```
          nx  = size x
          sx  = sub x (0, 1)
          xe  = select x 0
          xm  = sub x (1, nx)
          ny  = size y
          sy  = sub y (0, 1)
          ye  = select y 0
          ym  = sub y (1, ny)
```

```
ewp2a2v::(([Int], [Int])->([Int], [Int])) [Int] [Int]
           -> ([Int], [Int])
ewp2a2v f x y = ewp2a2v` f x y (empty, empty)
where
  ewp2a2v` f x y (r1, r2)
   | ((xs == 0) && (ys == 0)) = (r1, r2)
   | otherwise = ewp2a2v` f k l (concat r1 fx,
                                 concat r2 fy)
   where
     (sl, k, l) = slice x y
     (fx, fy) = f sl
     xs = size x
     ys = size y
```

The `combine` function is applied in the third phase to collect the results calculated from the blocks. `linfst` collects and concatenates the first components of the tuples from the input list, while `linsnd` does the same for the second components.

```
combine :: [([Int], [Int])] -> ([Int], [Int])
combine x = (linfst x, linsnd x)
```

The distributed computation is coordinated by D-Clean constructs. D-Clean coordination structures are mappings between communication channels of the process-network and are designed as generic templates parameterized by types and by functions. The value of type parameters are determined by type inference. The matching of types between the base types of channels and the types of embedded Clean expressions is a static semantic requirement. The templates are instantiated by the D-Clean pre-compiler at compile time.

A D-Clean program consists of a start expression, in which a collection of user-defined D-Clean process schemes can be applied. A process scheme itself is written in D-Clean too.

The start expression is given as the `DistrStart` function definition.

The Figure 1. illustrates the process network of the D-Clean implementation of the distributed elementwise processing.

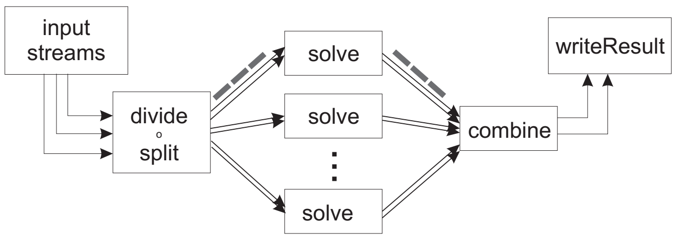

FIGURE 1. Elementwise processing in D-Clean

The task of `DStart` primitive is to start the distributed computation by producing the input data for the dataflow graph. It has no input channels, only output channels. Each D-Clean program contains at least one `DStart` primitive.

The `DEWP` scheme is used for coordinating the whole elementwise processing. It has three parameters: the elementwise processable function, the block size and the number of worker nodes.

`DStop` is the last element of the D-Clean composition, the last element of the control flow.

```
DistrStart = (DStop terminate) (DEWP union_inters bl N)
             (DStart generate)
  where
    N = 8
    terminate = WriteResult "c:\\dewp.dat" fromTup2
```

A D-Clean expression may be a compound expression or a direct use of coordination primitives. Process scheme definitions are named D-Clean expressions with formal parameters. A process scheme library can be built using the coordination primitives and the already defined schemes.

Here we give the description of the `DEWP` scheme implementing the elementwise processing for two arguments and two values.

```
SCHEME DEWP f bl N =
    DMerge (toTup2 (combine (fromTup2L)))
    DMap (toTup2 ((solve f) fromTup2))
    DDivideF (toTup (divide bl (fromTup2))) N
```

The newly introduces `DDivideF` distributes the blocks produced by the function `divide` to worker processes. The general form of `DDivideF` (see Figure 2) is `DDivideF exp N`. The first argument is an expression which produces a list of lists of values belonging to a transmissible-type. The number of sub-lists are usually greater than the number of worker processes defined by the constant value `N`.

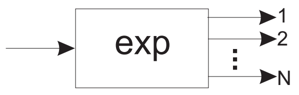

FIGURE 2. The `DDivideF` primitive

Every computation node receives a series of blocks. `DDivideF` implements a simple load-balancing algorithm. A worker can get a new block, if its input channel is empty. After sending the first set of sublists to the worker processes, `DDivideF` waits until one of the workers completes the processing of a block and reads in the next waiting block from the channel. Then this channel becomes empty and `DDivideF` places the next unprocessed block to this channel.

The `DMap` primitive coordinates the computation of the subresults. It applies the elementwise processable parameter function to every block at a separate node. `DMap` is a special case of `DApply` where the function expression must be an elementwise processable function. It is the D-Clean variant of the standard *map* library function.

The third phase collects the subresults, the computed values of the blocks. The `DMerge` combines the blocks forming the final output streams. `DMerge` collects the input sublists from channels and builds up the output data lists. All the input channels must have the same type

6 CONCLUSION

The distributed elementwise processsing example represents a computation pattern, which demonstrates the applicability of the introduced D-Clean coordination primitives. In our context the `DEWP` scheme serves as an easy to use computation skeleton in the distributed environment. It describes in general manner the common algorithmic scheme parameterized by elementwise processable function and data. The computation is distributed over the network according to the description provided by the skeleton.

This type of distributed computation is efficient due to the distribution of the same work over the network.

In our tests we measured and compared the performance of the sequential and distributed implementations. The implementations calculated the union and intersection of *integer* sets generated from a memory list, and saved to a disk file.

In the distributed test we used 7 worker processes with 4 additive nodes for generating, saving the result, and dividing and combining the subresults processed by the workers.

In order to simulate the performance of the parallel algorithm in case of more complex operations, we increased the calculation time of the `union_intersect` function.

The measured time of the sequential version was 2410 secs, while the time of the distributed version was 614 secs. This means a 3.92 speed-up.

REFERENCES

[1] Horváth Z., Hernyák Z, Zsók V.: Coordination Language for Distributed Clean, *Acta Cybernetica* 17 (2005), 247-271.

[2] Horváth Z., Zsók V., Serrarens, P., Plasmeijer, R.: Parallel Elementwise Processable Functions in Concurrent Clean, *Mathematical and Computer Modelling* 38 (2003), pp. 865-875, Elsevier.

[3] Cole, M.: Algorithmic Skeletons, In: Hammond, K., Michaelson, G., eds., *Research Directions in Parallel Functional Programming*, pp. 289-303, Springer-Verlag, 1999.

[4] Fóthi Á., Horváth Z., Kozsik T.: Parallel Elementwise Processing – A Novel Version, In: Varga L., ed., *Proceedings of the Fourth Symposium on Programming Languages and Software Tools*, Visegrád, Hungary, June 9-10, 1995, pp. 180-194. and in *Annales Uni. Sci. Budapest de R. Eötvös Nom. Sectio Computatorica*, Vol. 17, pp. 105-124, 1998.

[5] Galán, L.A., Pareja, C., Peña, R.: Functional Skeletons Generate Process Topologies in Eden, In: *Int. Symp. on Programming Languages, Implementations Logics and Programs PLILP'96*, Aachen, Germany, LNCS, Vol. 1140, pages 289-303, Springer-Verlag, 1996.

[6] Fóthi Á. et al., Workgroup on Relational Models of Programming - Some concepts of a Relational Model of Programming, In: Varga, L., ed., *Proc. of the Fourth Symp. on Programming Languages and Software Tools*, Visegrád, Hungary, June 8-14, 1995.

[7] Horváth Z.: Parallel asynchronous computation of the values of an associative function, *Acta Cybernetica*, Vol. 12, No. 1, pp. 83-94, Szeged, 1995.

[8] Darlington, J., Field, A.J., Harrison, P.G., Kelly, P.H.J., Sharp, D.W.N., Wu, Q., While, R.L., Parallel Programming Using Skeleton Functions, In: *Proc. PARLE '93 - Parallel Architectures and Languages Europe*, LNCS, Vol. 694, pp. 146-160, Springer-Verlag, 1993.

[9] Chandy, K.M., Misra, J.: *Parallel program design: a foundation*, Addison-Wesley, 1989.