Distributed Arrays in the Functional Language Concurrent Clean

Pascal R. Serrarens

Computer Science Institute
University of Nijmegen, The Netherlands
email: pascalrs@cs.kun.nl

Abstract. In this paper, we show how distributed arrays can be implemented in Concurrent Clean without extensions to the language. We introduce the notion of remote values, associating a value at a remote processor with its location. By combining remote values with arrays we can build distributed arrays, having the same flexibility as standard arrays. With the example of matrix-vector multiplication, we show that our distributed arrays are well suited to implement parallel algorithms with little overhead.

1 Introduction

At this moment, one can say that efficient sequential implementations of functional languages exist. Still, parallel functional programming is not very popular. Perhaps some features for effective parallel programming are still missing. In this paper we present an important construction for parallel programming in the functional language Concurrent Clean [PvE93], namely distributed arrays: arrays of which the elements are distributed over a number of processors. With Concurrent Clean, these arrays can be built efficiently without new language constructions. We continue with the work initiated by Kesseler [Kes95] and present a toolkit for working with distributed arrays.

In Section 2 we introduce remote values: values located on other processors. Section 3 then describes how distributed arrays can be built with remote values and gives some functions for them. In Section 4 we show an example of a matrix-vector multiplication on a mesh-shaped distributed memory computer model. Section 5 shows some results obtained with distributed arrays. Finally, Section 6 describes related work and Section 7 concludes the paper.

2 Remote Values

In Concurrent Clean it is possible to introduce parallelism by annotating expressions with the P-annotation [PvE93] [Kes94]. In the expression {| P at p |} f x, the expression f x is evaluated on the processor identified with p, where p has type ProcId. If the expression is not completely evaluated, the result is only partially returned to the root processor. The rest of the result is send when it is needed, which is not very efficient. In the rest of the paper we assume that all results are completely evaluated, so they are always send in one piece.

The P-annotation seems to be less suited for data-parallel programming because of two problems: first, it is not possible to determine the location of the result of the remote expression, and second, the result of the expression is communicated to the root processor very easily, as soon the remote expression is used, it gets evaluated and the result is send back to the root processor.

In his paper Kesseler [Kes95] used a new primitive function arg_id to solve the first problem: it returns the ProcId of its argument. For the second problem he used an ordinary datatype for representing remote values. We will also use such a datatype, but we add a new argument of type ProcId representing the location of the remote value:

```
:: R x = Remote x ProcId
```

With this datatype we have solved both problems: the location is stored with the value and can be determined easily. The Remote constructor protects the value x for being used directly.

We only have to make sure that the physical location of the remote value corresponds with the location stored in the datatype. This can be accomplished by making Remote an abstract datatype and providing a set of functions on it, each fulfilling the property above.

2.1 Moving Data and Information

The first three functions we introduce respectively copy data to, from and between processors:

```
putRemote :: ProcId a → R a
putRemote p x = Remote y p
where
    y = {| P at p |} x

getRemote :: (R a) → a
getRemote (Remote x p) = x

copyTo :: ProcId (R a) → R a
copyTo p2 (Remote x p1) = Remote y p2
where
    y = {| P at p2 |} x
```

The last function is important. Concurrent Clean provides implicit communication: when a value is needed at a different processor, the runtime system copies the value to the new location. But many parallel algorithms describe in detail how and when data is moved between processors, making sure that the communication overhead is as small as possible. Therefore, we provide a function for explicitly moving data between processors.

The last function is this section returns the location of a remote value. Together with copyTo, it can be used to align data on the same processor.

```
locationOf :: (R a) → ProcId
locationOf (Remote x p) = p
```

2.2 Sending Work

The key functions on remote values are the apply-like functions: they will send a function to the remote value and apply it there:

```
rap :: (a → b) (R a) → R b
rap f (Remote x p) = Remote y p
where
    y = {| P at p |} f x

rap2fst :: (a b → c) (R a) (R b) → R c
rap2fst f (Remote x p1) (Remote y p2) = Remote z p1
where
    z = {| P at p1 |} f x y
```

When applying a binary function, we have a choice where to leave the result: it may be the location of either argument, depending on the situation. Therefore we provide two functions: rap2fst and rap2snd which leave the result on the location of their first and second argument respectively.

3 Distributed Arrays

Using the remote values from the previous section we can build an array with elements located on remote processors. A distributed array is simply a type synonym for an array of remote values:

```
:: DArray a :== { R a }
```

Arrays in Clean are quite similar to lists, apart from some array-specific constructions. The type of arrays is denoted as { x }[1]. Clean provides ZF-expressions for manipulating arrays and list. The syntax of these expressions is adapted from mathematics for describing sets. Some examples of ZF-expressions are:

```
{ f e \\ e ← l }
{ g e1 e2 \\ e1 ←: a1 & e2 ←: a2 }
{ h e1 e2 \\ e1 ←: a1, e2 ←: a2 }
```

In the first example, the function f is applied to all elements of list l. In the second example the elements are drawn form arrays a1 and a2 simultaneously and g is applied to them. In third example is similar, but now the generators are nested. Each element of a1 is then combined with all elements of a2. Selecting an element from position i of array a is denoted as a.[i].

[1] In fact, Clean provides three kinds of arrays: lazy, strict and unboxed, with types { x }, { !x } and { #x } respectively [vG96]

3.1 Basic Operations on Distributed Arrays

Distributed arrays are created by combining two arrays: an array with the elements and an array for the locations of those elements. These two arrays are combined with putRemote:

```
createDArray :: { ProcId } { a } → DArray a
createDArray ps a = { putRemote p e \\ e ←: a & p ←: ps }
```

Many functions on distributed arrays are straightforward since they are basically array manipulations. We give the implementation of mapping a function over a distributed array and zipping two distributed arrays with a function.

```
map_Dist :: (a → b) (DArray a) → DArray b
map_Dist f dx = { rap f rx \\ rx ←: dx }

zipwithFst_Dist :: (a b → c) (DArray a) (DArray b) → DArray c
zipwithFst_Dist f dx dy = { rap2Fst f rx ry \\ rx ←: dx & ry ←: dy }
```

Similar to the functions rap2fst and rap2snd in Section 2, we have two zipwith versions: zipwithFst_Dist and zipwithSnd_Dist.

The current implementation of distributed arrays is not the most efficient. In case of the map_Dist function, the function f is applied to every element using rap, resulting in a message. When we have n elements in our array, a map will need n messages. In the future we plan a broadcast system to reduce this to a single message.

3.2 Rearranging Distributed Arrays and Communication

Kesseler [Kes95] did not use an explicit copy function to move data between processors. This was achieved by rearrangements of the array:

```
implRotateL_Dist :: Int (DArray a) → DArray a
implRotateL_Dist s dx = { dx.[(i + s) mod n] \\ i ← [0 .. n - 1] }
where
    n = size dx
```

The advantage of this is that it saves communication: when two rotations are performed after each other, the elements are copied only once. However, it may also lead to too little communication. Take for example the following expression:

```
map_Dist ((+) 1) (implRotateL_Dist 1 dx)
```

The rotation suggests that communication takes place, but the map function uses the locations stored in the distributed array, which did not change by the rotation. Therefore, the rotation will not be performed. This problem may be very hard to detect. When the replacement of subarrays is explicit, we do not have this problem. The subarrays are copied explicitly with the copyTo function:

```
rotateL_Dist :: Int (DArray a) → DArray a
rotateL_Dist s dx =
    { copyTo (locationOf dx.[i]) dx.[(i + s) mod n] \\ i ← [0 .. n - 1] }
where
    n = size dx
```

3.3 Folding Distributed Arrays

Fold-like functions combine the elements of compound datastructures using a binary function. The best choice for a parallel fold is a tree shaped fold, which make an effective use of parallelism. However, the location of the result is not determined and the communication distances will not be minimal. An option is are left-to-right-fold, which behaves like a left-fold and leaves the result at the same location of the rightmost element. Its complement is right-to-left-fold, similar to a right-fold and leaving the result at the leftmost location.

```
foldlr_Dist :: (a a → a) (DArray a) → R a
foldlr_Dist f dx = foldlr 1 dx.[0]
where
    foldlr i r
        | i >= size dx = r
        | otherwise    = foldlr (i + 1) (rap2snd f dx.[i] r)
```

4 An Example: Matrix-Vector Multiplication

In this section we will implement the matrix-vector multiplication algorithm by Kumar et al. [KGGK94]. The algorithm assumes a mesh-shaped processor network. The matrix is distributed blockwise over the network, while the vector is distributed blockwise over the rightmost column of the network. In the algorithm, the vector is first distributed over the processors, then all processors perform a local matrix-vector multiplication and at the end, the local results are summed, leaving the result vector in the rightmost column of the processor network.

We need two kinds of distributed structures: a distributed matrix and a distributed vector, with types:

```
:: DMatrix :== {{ R Matrix }}
:: DVector :== { R Vector }
```

For the broadcast of the vector, the algorithm prescribes a two-step mechanism, to get a minimum communication overhead. In the the first step the vector is copied to the diagonal of the network:

```
copyToDiagonal :: DVector DMatrix → DVector
copyToDiagonal dv dm =
    { copyTo (location of dm.[i,i]) dv.[i] \\ i ← [0 .. size dv - 1] }
```

Table 1. Performance measurements for the matrix-vector multiplication. t = running time in seconds, s = speedup compared to sequential code

matrix size	sequential	4 processors		16 processors		64 processors	
	t	t	s	t	s	t	s
512^2	6.2	1.7	3.7	0.7	8.8	1.2	5.2
2048^2	97.6	24.7	3.9	6.7	14.6	3.0	33.0
8192^2	1555.5	390.1	4.0	98.9	15.7	27.1	57.5

In the second step the vector is broadcasted over the columns. The result of this step is a distributed vector for each row of processors:

```
distributeVector :: DVector DMatrix → { DVector }
distributeVector dv dm =
    {{ copyTo (locationOf dm.[r,c]) dv.[c] \\ c ← [0..nc - 1] } \\ r ← [0..nr - 1] }
where
    nr = size dm
    nc = size dm.[0]
```

Next, every processor evaluates its own local matrix-vector multiplication. For that we use zipWithFst_2Dist, a 2-dimensional variant of zipWithFst_Dist:

```
localMatMulVec :: DMatrix { DVector } → { DVector }
localMatMulVec dm dv = zipWithFst_2Dist matVecMult dm dv
```

Every row of the distributed vector is summed from left to right, with the result ending up in the rightmost column. We do this for every row in the array. We assume that the addition operator (+) is defined for vectors:

```
accumSums :: { DVector } → DVector
accumSums dv = { foldr1_Dist (+) row \\ row ←: dv }
```

5 Performance Measurements

Our implementation of the matrix-vector multiplication was tested on a 64-node transputer network. We compared the code against sequential code with no overheads for parallelism. Table 1 shows good speedups, because the local matrix-vector multiplications take most of the time.

Another test case the conjugate gradient algorithm. The sequential version compared well to C and Haskell [Ser96]. The parallel implementation ran on a network of 4 Macintosh II computers connected by Localtalk and Ethernet. Using many map-like operations, we had lots of communication, so worse speedups could be expected, but table 2 shows that the results were encouraging. We

Table 2. Performance measurements for the conjugate gradient algorithm.

matrix size	sequential	4 Localtalk		4 Ethernet	
	t	t	s	t	s
400^2	22.1	67.9	0.3	25.5	0.9
1600^2	175.9	152.7	1.2	83.1	2.1
3600^2	590.9	306.9	1.9	200.7	2.9
6400^2	1386.9	552.7	2.5	416.1	3.3

expect that these times will improve when broadcasting as described in section 3.1 is implemented.

The performance for matrix multiplication is close to the results obtained by Kesseler. For multiplication of a matrix of size 256×256, the running times in seconds on 4 and 16 processors were 4.0 and 13.3 for our implementation, compared to 4.0 and 15.5 for Kesseler. Our times are slightly better, because the explicit movement of data gives less parallel overhead.

6 Related Work

The remote values in this paper are similar to network objects [BNOW94]. A network object consists of a local and a remote object. Invocation of a method of the local object will send a message to the remote object which then invokes its corresponding method. Network objects do not differ from normal objects for the programmer, while we need the special apply function rap for remote values.

Various other functional languages provide some kind of distributed array, but they all provide it as a primitive language construction. The primitive parallel datatype of NESL [BCH+93] are sequences. It uses a construction similar to ZF-expressions for the application of a function over a sequence.

Skil [BK96] tries to have maximal flexibility with its pardata construct. It allows any distributed data structure, as long as it is composed of identical data structures place on each processor. Nested parallelism is not possible.

SCL [DGTY95] give the ParArray for data-parallelism. It also provides configuration skeletons [Col89] to determine the distribution and alignment of the data. No direct control over locations of values is possible in NESL and Skil.

Kuchen and Geiler [KG91] suggest distributed arrays in favor of list- and tree-like structures. Every processor has an array with links to non-local distributed arrays elements. In this way processors have easy access to array elements at other processors, but is more complex to manage than our solution.

7 Conclusions and Future Work

In this paper we showed how an efficient implementation of remote values can be implemented in Concurrent Clean. Apart from the standard P-annotation, we did not need any other extension of the language. Remote values are used to build and efficient implementation distributed arrays, an important construction for parallel computing. We showed that good speedups are possible with our distributed arrays.

We will continue improving the implementation of distributed arrays, with emphasis on minimal communication costs. This will include broadcasting functions and values over the network. The possibility of using remote values in other datastructures will also be a topic of further research.

References

[BCH+93] G.E. Blelloch, S. Chatterjee, J.C. Hardiwck, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Principles and Practice of Parallel Programming (PPoPP)*, pages 102–101, 1993.

[BK96] G.H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Proceedings of the Fifth International Symposium on High Performance Distributed Computing (HPDC-5)*, pages 243–252. IEEE Computer Society Press, 1996.

[BNOW94] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. Technical Report Digital SRC 115, DEC, Palo Alto, USA, 1994.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989.

[DGTY95] J. Darlington, Y. Guo, H.W. To, and J. Yang. Functional skeletons for parallal coordination. In *Proceedings of Europar 95*, 1995.

[Kes94] M.H.G Kesseler. Uniqueness and lazy graph copying - copyright for the unique. In *Proceedings of the 6th International Workshop on the Implementation of Functional Languages*, Norwich, UK, 1994. University of East Anglia.

[Kes95] M.H.G Kesseler. Constructing skeletons in clean - the bare bones. In *Proceedings of High Performance Functional Computing (HPFC '95)*, pages 182–192, Denver, Colorado, 1995. Lawrence Livermore National Laboratory. CONF-9504126.

[KG91] H. Kuchen and G. Geiler. Distributed applicative arrays. Technical Report AIB 91-5, RWTH Aachen, 1991.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing, Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc., California, 1994.

[PvE93] M.J. Plasmeijer and M.C.J.D. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley Publishers Ltd., 1993.

[Ser96] P.R. Serrarens. A clean conjugate gradient algorithm. In *Proceedings of the 8th International Workshop on Implementation of Functional Languages*, pages 367–376, Kiel, Germany, September 1996.

[vG96] J.H.G. van Groningen. The implementation and efficiency of arrays in Clean 1.1. In *Proceedings of the 8th International Workshop on Implementation of Functional Languages*, pages 131–154, Kiel, Germany, September 1996.

This article was processed using the LaTeX macro package with LLNCS style