Functional Multicasting

Pascal R. Serrarens

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

Abstract. Multicasting, where a message is sent from one processor to number of other processors, can optimise the communication for data-parallel functional computing. This paper describes the introduction of two multicast primitives in the functional language Concurrent Clean. The usage and performance are showed, with special emphasis on the usage in combination with distributed arrays.

In a data-parallel programming model, many instructions are sent to all processors. An efficient way to do this is broadcasting, which sends a message simultaneously to all processors available. Some interconnection schemes, like buses and trees, can support broadcasting directly, because the sender can have a direct connection to all other processors. Distributed architectures based on other interconnection schemes often have special hardware to support broadcasting. Sometimes a variant of broadcasting, multicasting is supported, where a message is sent only to a limited subset of processors. Broadcasting is a real optimisation technique: it does not introduce new functionality to a system, because the same effect, albeit with lower performance, can be achieved by sending the same message multiple times.

There has been a lot of work on parallel functional programming, but broadcasting or multicasting has not received much attention. This is partly because many ways of introducing parallelism in functional programs are not suited for it. Annotations [PvE93] annotate a function application which will be send to one processor only. The `par` primitive of GpH [PH93] evaluates one expression at one processor. Strategies [TLHP96] could make use of multicasting, but no work on that has been mentioned yet. Other ways for introducing parallelism, like skeletons [Col89] are rather high-level and an issue like multicasting is considered to be part of the implementation of the runtime system and not part of the language.

In this paper we will introduce two multicast primitives which use the broadcast or multicast hardware to achieve better performance. No new functionality is added, as we can describe the semantics in standard Clean. The main use for multicasting is optimising the implementation of distributed arrays [Ser97].

1 Using multicasting

A good example of a data-parallel function where broadcasting can be used is the parallel map. It can be defined using Clean's P-annotation and array comprehensions. Array comprehensions are similar to list comprehensions, they are enclosed by curly brackets { }, instead of square brackets [ ]. The symbol ←: is used to draw an element from an array, while lists use ←.

```
par_map :: (a → b) { a } → { b }
par_map f a = {  {| P |} f e \\ e ←: a }
```

In this implementation n expressions will be send to remote processors, namely f a.[0], f a.[1], f a.[2] ... f a.[n - 1] (Selecting the i-th element of array a can be written as a.[i]). Each of these expressions will be packed and send separately, which gives a relative large communication overhead. Alternatively we could send the function f with the complete array a to all processors simultaneously. Each process will then select the correct element from the array before it applies f to it.

We will introduce a new primitive function which uses the broadcast hardware available on many distributed computers to evaluate the elements of an array in parallel more efficiently. This will give us the possibility to implement faster versions of the parallel map and similar functions. The new primitive is not a real broadcast, but a multicast, because it sends the expression to a limited set of processors. The set of processors which should evaluate the elements of the result array, is represented as an array of processor identifiers (ProcId's) and are an argument of the function.

The problem with broadcasts and multicasts is that every processor receives the same expression. Evaluating the same expression on all processors is not very sensible, except in safety-critical systems, where hardware errors can be detected by replicating tasks.

If we assume that every process knows the index value of the element it should evaluate, we can have different behaviour for every task as the value is different for every process. An element of the result array can be evaluated by applying a certain function g to its index value. In the case of the parallel map, g could be:

```
g :: Int → a
g i = f a.[i]
```

The index value for each process, can be deduced from the array with process identifiers determining the receiving set of processors. If, for a certain processor p, the i-th element of the array with processor identifiers equals p, it should apply the function g to i. So when the array processor identifiers equals $\{P_0, P_1, P_0\}$, processor 0 will create processes evaluating g 0 and g 2, and processor 1 will evaluate g 1.

The remote expressions will be represented by channels in the local heap. When the result of a remote expression is needed, the channel will make sure that it will be copied to the right processor. The number of processors available is always finite, so the channels can be well stored in an array. It may be wise to use a lazy array [vG96], which has type { a }. With a strict or unboxed array, with types { !a } and { #a } respectively, the elements are always transported to the local processor, while an element of the result array may not be needed or may be needed on a different processor.

Our new multicast function has the following type:

```
multicast :: { # ProcId } (Int → a) → { a }
```

The semantics of multicast, ignoring the use of broadcast hardware, can be described by using Clean's P-at-annotation and array comprehensions:

```
multicast ps f = {  {| P at ps.[i] |} f i \\ i ← [0 .. size ps - 1]  }
```

Now we can implement a more efficient version of the parallel map. In this example the functions are distributed randomly over the processors. The function randomP returns a random ProcId, but cannot influence the function result, as ProcId's cannot appear in the program result.

```
par_map :: (a → b) { a } → { b }
par_map f a = multicast { randomP \\ i ← [0 .. size a - 1] } g
where
    g i = f a.[i]
```

Another example where multicasting can be used is a parallel version of the make_array function. It is given a generator function and the length of the new array, evaluates the elements in parallel and returns an array with channels to the elements:

```
par_makearray :: (Int → a) Int → { a }
par_makearray f n = multicast { randomP \\ i ← [0 .. n - 1] } f
```

2 Updating Multicast

The multicast primitive from the previous section has one drawback: it always creates a new array for the result. This is not always necessary: you could have an array which can be recycled, using update-in-place. Therefore we give a alternative multicasting primitive, which does exactly that:

```
u_multicast :: { #ProcId } (a Int → a) *{ a } → *{ a }
```

It takes an extra argument, a unique lazy array, which can be updated safely, because the uniqueness typing system ensures that the array will not be shared [BS93]. The function passed to u_multicast now also takes the element at each position, because when a selection is used, as in par_map above, the array will loose its unicity. Like the multicast primitive, we can define the semantics, ignoring broadcast hardware, with a normal Clean function: (When two generators are separated by &, the elements are drawn simultaneously. In an array comprehension the construction { a & [i] = ...} means that it will update array a at position i)

```
u_multicast pids f a = { a & [i] = {| P at ps.[i] |} f e i \\ e <-: a & i <- [0 ..] }
```

Below, an example is given of an updating map, which reuses the array argument:

```
par_map_u :: (a -> a) *{ a } -> *{ a }
par_map_u f a = u_multicast { randomP \\ e <-: a } g a
where
    g e i = f e
```

It could be possible to keep u_multicast as the only new primitive function, because the function multicast can be defined as:

```
multicast ps f = u_multicast ps g (createArray (size ps) (f 0))
where
    g e i = f i
```

This implementation of multicast can be much more inefficient than the primitive. The problem is that, in the code above, the array needs to be initialised with the result of the expression (f 0), which may be expensive to reduce. The primitive multicast doesn't need the initialisation, but fills the array immediately with the channels to the elements. So for efficiency reasons we introduce both multicast primitives.

3 Distributed Arrays

The arrays returned by the functions above are not very convenient. First, the locations of the elements are not clear. Second, the lazy evaluation scheme of Clean makes it hard to reason when a channel is needed. This results in communication where it is not expected or needed.

To alleviate this problem remote values were introduced [Ser97]: they combine a value with its location, while the constructor of the datatype prevents unwanted communication [Kes95]. The datatype is defined as:

```
:: R a = Remote a ProcId
```

It was shown that an efficient implementation of distributed arrays can be made with standard Clean, using an array of remote values. Multicasting could improve those results, because many times, the same function is sent to the processors.

To use multicasting to it's best, we introduce an alternative implementation for distributed arrays. Instead of using an array of remote values, we make the new datatype:

```
:: DArray a = DA { #ProcId } { a }
```

Again, we use a ordinary constructor to prevent unnecessary communication. The first array argument of the datatype describes the locations of the elements of the second.

The initial data distribution can be achieved using a multicast:

```
distribute_Dist :: { #ProcId } { a } → DArray a
distribute_Dist ps a = DA ps (multicast ps f)
where
    f i = a.[i]
```

Map-like functions like map and zipwith are straightforward, because they send the same function the processors:

```
map_Dist :: (a → b) (DArray a) → DArray b
map_Dist f (DA ps a) = DA ps (multicast ps g)
where
    g i = f a.[i]
```

Another, less obvious, example is a tree-shaped fold on arrays. It uses a multicast for every level in the tree.

```
fold_Dist :: (a a → a) (DArray a) → R a
fold_Dist f (DA ps a)
    | size a == 1            = rselect_Dist (DA ps a) 0
    | (size a) mod 2 == 0 = fold_Dist f (DA ps' (multicast ps' g))
    | otherwise              = fold_Dist f (DA ps' (multicast ps' h))
where
    ps' = { ps.[i] \\ i ← [0, 2 .. size ps - 1 ] }

    g i = f a.[i * 2] a.[i * 2 + 1]

    h 0 = a.[0]
    h i  = f a.[i * 2 - 1] a.[i * 2]
```

For some other operations, we should have access to the individual elements of the distributed array. Therefore, we give the function rselect_Dist, which selects an element from an array and returns it as a remote value. Updating one element of the array can be done with ap_Dist, which doesn't use remote values, but updates the distributed array in-place with a channel to the new element being evaluated by a new process. (The construction a=:{[i]=e} is like a pattern match on an array: the i-th element of array a is selected and assigned to e. When the array a is not needed, the first part can be left out, as in rselect_Dist.)

```
rselect_Dist :: (DArray a) Int → R a
rselect_Dist ( DA {[i]=p} {[i]=e} ) i = Remote e p
```

```
ap_Dist :: (a → a) *(DArray a) Int → *DArray a
ap_Dist f ( DA ps a=:{[i]=e} ) i = DA ps (update a i e‘)
where
    e‘ = {| P at ps.[i] |} f e
```

The rselect_Dist function is convenient when operations should be applied on individual elements, as example the foldlr_Dist from [Ser97] using rselect_Dist is given.

```
foldlr_Dist :: (a a → a) (DArray a) → R a
foldlr_Dist f dx = foldlr 1 (rselect_Dist dx 0)
where
    foldlr i r
        | i >= size_Dist dx = r
        | otherwise          = foldlr (i + 1) (rap2Snd f r (rselect_Dist dx i))
```

4 Performance measurements

The multicast primitive has been included in a distributed implementation for a network of Macintosh computers. It involved a major redesign of the communication mechanism, because the Appletalk ADSP point-to-point protocol we were using did not support broadcasting. The Appletalk DDP protocol supporting broadcast is more low-level and gave us the opportunity to cut down the overhead and therefore the communication costs.

We did some initial testing on up to four Apple Macintosh II computers connected by Ethernet. We tested a parallel implementation of the conjugate gradient algorithm [KGGK94], which uses a large number of map-like functions, so good speedups could be expected. For problem size 12×12 we obtained the following results:

	sequential	2 procs	3 procs	4 procs
ADSP	4.50	6.81	10.00	13.35
DDP	4.50	6.61	8.68	10.90
DDP + multicast	4.50	8.05	7.28	7.61

Table 1. Execution time in seconds for pure sequential and parallel implementations of the conjugate gradient, problem size = 12×12.

Due to problems with the garbage collector, we were not able to test larger problems and did not get real speedups compared to the pure sequential version. This example, however, already shows that multicasting can improve the performance considerably, especially on larger networks. On a two-processor network, we actually had a speed-down, because the multicast primitive gives more overhead than a P-annotated function.

5 Future work

Multicasting can also be used to improve the performance when answering requests to channels. When a channel on one processor is copied to a group of processors, it is likely that the processor holding the data will receive multiple requests. When that data has been reduced to root normal form, it can be send to all processors waiting for the data simultaneously using a multicast.

In Clean, the I- and P- annotations were the only way to introduce parallelism. They cannot change the outcome of a program, but change the place where a expression is evaluated. Therefore, if they are left out, a normal sequential program remains. The primitives introduced in this paper cannot be left out easily because they are ordinary functions. An alternative for those primitives is a new annotation, which is only allowed on array comprehensions. The annotation should not be allowed on list comprehensions, because for a multicast, the result structure should be finite. The elements of the result array of the annotated array comprehension will be evaluated on the indicated processors. A parallel map, using the annotation, could look like:

```
y = {| PS at pids |} { f x \\ x ←: xs }
```

If the annotation is left out of the function above, we have a normal map over arrays.

An annotated array comprehension will be transformed using different rules than the normal set and uses the multicast and u_multicast primitives introduced in this paper to distribute the expressions over the processors. The two primitive functions should become system functions, which are not available for the programmer, as it is possible to achieve the same with the annotated array comprehensions:

```
multicast pids f = {| PS at pids |} { f i \\ i ← [0 .. size pids - 1] }

u_multicast pids f a = {| PS at pids |} { a & [i] = f i \\ i ← [0 .. size a - 1] }
```

6 Conclusion

With the introduction of multicasting in Clean, a more efficient implementation of important parallel constructions can be made. We showed that the proposed primitives are referential transparent and and particularly suited for data-parallel programming. The initial speedups we achieved were up to a factor 2 compared to the old implementation of the communication layer and 30% faster performance than with the new communication routines without using multicasting. Multicasting seems to be most important when a program runs on a network with 3 or more processors. In a small network the performance could go down.

References

[BS93] E. Barendsen and J.E.W Smetsers. Conventional and uniqueness typing in graph rewrite systems. In R.K. Shyamasundar, editor, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *LNCS*, pages 41–51. Springer-Verlag, 1993. extented abstract.

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

[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.

[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.

[PH93] J.C. Peterson and K. Hammond. Report on the non-strict functional language Haskell, version 1.3. Technical report, University of Glasgow, 1993.

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

[Ser97] P.R. Serrarens. Distributed arrays in Clean. In *Proceedings Euro-Par'97*, LNCS, 1997.

[TLHP96] P.W. Trinder, H.-W. Liodl, K. Hammond, and S.L. Peyton Jones. Algorithm + strategy = parallelism. In *Proceedings of the 8th International Workshop on Implementation of Functional Languages*, pages 377–395, 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