Parallel Functional Programming with Bags

Giuseppe A. Manno and Giancarlo Succi

DIST Università di Genova
via Opera Pia 11a
I-16145 Genova
Italy

1 Forms of Parallelism

One of the biggest appeals of functional programming languages is what is commonly referred to as “Implicit Parallelism”, i.e., the parallelism a compiler can automatically and easily identify and exploit, since no explicit constraint on the execution order is imposed in the abstract logic evaluation scheme. Several attempts have been made by using this property in order to design and implement a language which can fully exploit the potentiality of the (parallel) architecture of its target machine, possibly without resorting to ad hoc constructs and/or annotations.

Various forms of parallelism have been evidenced, nevertheless the results that have been achieved are not as expected. Probably the initial goals were set too high, but anyway we claim that too much attention has been devoted to the most “expensive” form of parallelism, expensive in terms of time and space overheard required for communications, synchronizations and processes management. On the other side, very little attention has been given to other simple forms of intrinsic parallelism, which could have clearer and simpler definitions and can be used to more straightforward implementations.

We can divide in two classes the intrinsic parallelism of functional programs: processes parallelism and data parallelism. Process parallelism is the form of parallelism which parallelizes the execution of independent parts of a program, while data parallelism, which is almost unexplored within the field of functional programming, consists of applying in a data parallel fashion operations on large collections of objects.

It is evident that process parallelism have some intrinsic limitations since it requires quite a lot of communications between processes for exchanging data and synchronizing a lot of time for forking and joining processes and a lot of bookkeeping to have a consistent execution. Furthermore there is the not decidable problem of how to choose what processes should run on what processors. On the other side, data parallelism seems quite appropriate for an implementation on both MIMD parallel architectures and the SIMD ones, like the Connection Machine and it overcomes most of the limitations of process parallelism. The question is then which kind of collections we should use and for this purpose we propose the unordered collection of objects or bag.
2 An Axiomatic Definition of the Bag

For the sake of clarity, we start with a rather intuitive definition of bag. In fact, we will change it in the next section into a new one which is more suited for a parallel environment.

Let $T$ be a type, $D$ the domain of the object of that type; furthermore, let $D_1$ be $D \cup \{1\}$ and $B(T)$ the domain of the bags of objects of type $T$. Now we can define the bag as:

$$B(T)=\langle T, \{\text{emptybag, any, add, sub, bmember}\} \rangle$$

Where the primitives have the following types (using curried funct):

- emptybag :: $D_1$
- any :: $B(T) \rightarrow D_1$
- add :: $B(T) \rightarrow D \rightarrow B(T)$
- sub :: $B(T) \rightarrow D \rightarrow B(T)$
- bmember :: $B(T) \rightarrow D \rightarrow \text{Bool}$

And obey the following axioms ($a$ and $b$ are different elements of $D$, $s$, $s_1$ and $s_2$ as elements of $B(T)$):

1. Construction

   $$( s_1 = \text{add} a ( \text{add} b s_1 ) ) \land ( s_2 = \text{add} b ( \text{add} a s_1 ) ) \rightarrow ( s_1 = s_2 )$$

2. Selection

   $$(\text{any emptybag} = 1) \land ( a = \text{any} s ) \rightarrow (\exists s_1 : s = \text{add} a s_1 )$$

3. Subtraction

   $$((\forall s_1 : ( a \neq \text{add} a s_1 )) \rightarrow (\text{sub} a s = s_1 ))$$

4. Membership

   $$\text{bmember} a s \rightarrow (\exists s_1 : ( s = \text{add} a s_1 ))$$

For further detail see [MS91].

3 Toward a Parallel Implementation

3.1 Planning for Parallelism

Inssofar, we have described a rather unconventional data structure, taking care of highlighting its intrinsic non-deterministic behavior. Now we present how we can efficiently design it: a straight implementation of this data structure, whereas is always correct under a semantic point of view and satisfies our first two aims (no unnecessary overhead for construction and access), falls short in the third requirement (easy to parallelize), hence we thought at a different one, semantically equivalent to the previous (despite being much less intuitive) but much more tailored to a parallel implementation. Here below is shown that the new primitives can be rewritten in terms of the old ones and it can also be proven the vice versa. The examples we present in this paper are based on these, more effective, primitives.

Instead of the definition given above, we can say that the bag is:

$$B(T)=\langle T, \{\text{emptybag, any, add, bdistr, bfilter, bfold, bunion}\} \rangle$$

fold :: \[(a\rightarrow a) \rightarrow a \rightarrow [a] \rightarrow a\]  
fold f z [] = z  
fold f z (a:x) = f a (fold f z x)

efold :: \[(a\rightarrow a) \rightarrow a \rightarrow \text{bag} a \rightarrow a\]  
efold f z s = f (bfold f z s) (bfold f z s2)

where

$$(s_1, s_2) = \text{cut} s$$

$$(c_1 = \text{bunion a b1}) \land (c_2 = \text{bunion a b2}) \rightarrow (c_1 = c_2)$$

(\text{bdistr} f b1) \land (c_2 = \text{bfilter p b2}) \land (b1 = b2) \rightarrow (c_1 = c_2)$$

Although $\text{bfold}$ and any are intrinsically non deterministic, it has been shown [MS98] that $\text{bfold} f$ behaves as a proper function for all $f$ commutative and associative. Thus, for all practical programming applications, it is possible to apply the usual methods of correctness analysis, program transformations, etc., by means of the usual mathematical tools valid for functional programming languages. We are not facing any problems of fairness, because we are dealing only with bags of finite size.

### References