A New Approach to Parallel Functional Programming

Giuseppe A. Marino
DIST
Università di Genova
I-16145 Genova, Italia

Giancarlo Succi
DIST
Università di Genova
I-16123 Genova, Italia

Abstract

Functional languages exhibit “Implicit Parallelism” which has been used in the attempt to achieve an efficient execution on parallel architectures; as a matter of the fact, the work has been focused on what we call process parallelism, leaving almost unexplored the field of data parallelism despite it seems quite promising. To exploit properly data parallelism it is necessary to design a suitable collection. For this aim we introduce a new data structure, the bag, and we give an intuitive explanation, a formal definition and some examples of it, evidencing its non-deterministic aspects and showing its impact on an effective implementation on a parallel architecture. Afterwards some practical justification of the usage of bags are presented. Finally we discuss some experimental results, which are remarkably encouraging, and we draw our conclusions.

1 Introduction: What Kind 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 posed in the abstract logic evaluation scheme. Several attempts have been made for using this property in order to design and implement a language which can fully exploit the potentialities 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 overhead 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 due to more straightforward implementations.

In this paper we analyze a kind which we call data parallelism: section 2 is focused on its definition; section 3 raises the issue of unnecessary constraints we often have on the data structure we use; section 4 introduces the bag, a data structure without most of those constraints; section 5 describes how the bag can be tailored for a data parallel execution; section 6 gives some examples of applications of the bag; section 7 explains how we measured the performances; section 8 presents the results we have been obtaining and discusses them; section 9 outlines some open questions and draws some conclusions.

2 Data Parallel Functional Programming

We can divide in two classes the intrinsic parallelism of functional programs:

- process parallelism,
- data parallelism.

Process parallelism is the form of parallelism which parallelizes the execution of independent parts of a program, usually with the following approach:

1. identify closures which can be executed independently,
2. decide whether to execute them or not,
3. schedule the ones we have decide to execute on the processors,
4. collect the results.

It is evident that this approach 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.

A different design can be devised and a good candidate for it seems data parallelism, which is almost unexplored within the field of functional programming. It can be described by the following approach:

1. identify the collections of objects globally handled by our program,
2. decide how to spawn them onto the available processors,
3. manipulate them applying, as much in parallel as possible:
   - “element-wise” operators,
   - filters,
   - “folding” operators;
in order to obtain either new collections or just scalar elements.

This approach 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 how we should spawn the collections onto the available processors.

3 Collections in Functional Programming

Several built-in collections are proposed by researchers for functional languages, such as sequences introduced by Backus[1], vectors, especially suited for the Connection Machine [3], sets[5, 9], but nevertheless the most common one is still the list, firstly used in the very early Lisp in the late Fifties as an abstraction of the Turing Machine tape. The reasons for its success are several: it is conceptually simple, it can be easily and efficiently implemented and so on; however a broad usage of lists carries some disadvantages which can be understood analyzing its usual definition:

- nil is a list,
- if 1 is a list and a is an object, then cons(a, l) is a list.

i.e., it represents a multiset with a total order over its elements. This happens also when a total order is meaningless: in this case an arbitrary order (based on the order of evaluation of cons) is implied. When dealing with unordered collections this extra constraint is responsible for the \(\Theta(n)\) (worst case) time overhead\(^1\) on each member-wise operation, for instance consider the widely used “apply-to-all” higher order function map:

\[
\text{map } f [] = [] \\
\text{map } f (a:l) = f a : \text{map } f l
\]

Furthermore, it causes an implicit bottleneck on parallel implementations which could be alleviated only by means of complex optimization techniques.

Therefore a data structure that suits a data-parallel framework should obey the following three requirements:

1. to be intrinsically simple to implement, i.e., no costly operations (such as duplicate check) are needed,
2. to eliminate unnecessary constraints (such as the linear access time to the elements),
3. to be especially tailored for parallelism, with an emphasis on distributed memory systems.

The set and multiset data structures could be considered good candidates. Although a true set data structure could be more attractive from an intellectual point of view, to implement sets consistently with set-theory we need to perform the check for duplicates which has a worst case quadratic time cost in the cardinality of the set.

We claim that the multiset, or bag, can be seen as the least-constraining collection. As such it has the following good characteristics which match quite well the previously mentioned requirements:

1. it is intrinsically simple (it can be implemented avoiding costly operations),
2. it does not impose unnecessary constraints (like the total ordering),
3. it can exploit parallelism of algorithms (especially with reference to distributed memory architectures).

4 An Axiomatic Definition of the Bag

In this section we give a simple formal definition of bag taking a simple axiomatic approach: first we define a bag in terms of the primitives which operate on it, then we give the types of the primitives and eventually we present the axioms which should be satisfied by them.

We use the word primitives instead of functions since they sometimes present nondeterminism: our underlying belief is that the non determinism we introduce here can offer great deal of opportunities for par-

\(^1\)Where 'n' is the cardinality of the data structure.
allelism, as we will describe in the following sections. We will not go into the theoretical properties of non
determinism since they go far beyond our subject here.

For the sake of clarity, we start with a rather intu-
itive definition of bag, then 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 $D_\bot$ be $D \cup \{\bot\}$ and $B(T)$ the
domain of the bags of objects of type $T$. Now we can
define the bag $B$ as:

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

Where the primitives have the following types (using
curried functions notation):

- emptybag :: $B(T)$
- any :: $B(T) \rightarrow D_\bot$
- 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, s1 and s2 are elements of $B(T)$):

1. Construction
   
   $$(s1 = \text{add } a (\text{add } b \ s)) \land
   (s2 = \text{add } b (\text{add } a \ s)) \rightarrow (s1 = s2)$$

2. Selection
   
   $$(\text{any emptybag} = \bot) \land
   (a = \text{any } s) \rightarrow (\exists s1: \ s = \text{add } a \ s1)$$

3. Subtraction
   
   $$((\exists s1: (s = \text{add } a \ s1)) \rightarrow
   \text{sub } a \ s = s1) \lor
   ((\forall s1: (s \neq \text{add } a \ s1)) \rightarrow
   \text{sub } a \ s = s)$$

4. Membership
   
   $$\text{bmember } a \ s \leftarrow (\exists s1: (s = \text{add } a \ s1))$$

Axiom (1) above defines the lack of total order with-
in the bag container. If the number of elements of a
bag is more than 1, they are more than one
“add-expressions” defining it: any permutation of its
elements yields a valid expression, e.g.,

$$s = \text{add } e1 (\text{add } e2 (\ldots \ldots \ en)\ldots ) = \text{add } e2 (\text{add } e1 (\ldots \ldots \ldots )) = \ldots = \text{add } en (\ldots \ldots e1)\ldots )$$

Axiom (2) defines the non-determinism in selection;
nothing must be inferred about which element is re-
turned by an application of any since it is not a “prop-
er” function, therefore:

$$\{ \begin{array}{l}
x = \text{any } b \\
y = \text{any } b \neq x = y
\end{array}$$

This axiom means that if $a = \text{any } s$ then there exists
another bag $s1$ so that $s$ can be obtained adding $a$ to
$s1$.

Axioms (3) and (4) are based on the equivalence
of all bags made up of the same elements, as stated
above. Axiom (3) says that if $s1 = \text{sub } a \ s$ then
if $s$ contains $a$, $s1$ is obtained subtracting $a$ from $s$,
otherwise $s1 = s$. Axiom (4) represents a constructive
approach to the membership test.

5 Toward a Parallel Implementation

5.1 Planning for Parallelism

A straight implementation of this data structure,
whereas is truly 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. It
can be 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:

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

Informally, bdistr applies a given function to all the
elements of the bag, while bfilter selects the elements
of a bag satisfying a given predicate, bfold reduces a
bag to a single object repeatedly applying a binary
function, and bunion produces the union of two bags.

bdistr, bfilter and bfold are not completely o-
riginal, all the languages that allow higher-order func-
tions have operators (either built-in or constructed)
that resembles the behavior of these primitives on var-
ious data structures: FP has $\alpha$ (apply) and $/ \beta$ (insert)
over sequences, CMLisp defines $\alpha$- and $\beta$-functions
over vectors, SASL and Miranda define map and fold
over lists. Our proposal owes some credits to the SASL
7.2 Our System

The bag data abstraction and its basic operators have been inserted in an existing SASL[8] compiler and interpreter[7]. The bag is represented in the underlying imperative language as a bit vector whose size is the maximum number of allocable cells of the abstract machine: in this way we would be able to easily implement this language on a SIMD parallel architecture where a bag may be represented by a long word with one bit per PE. The basic functions manipulate bags in a (simulates) parallel fashion, and the user of the interpreter can choose how many PEs the compilation is spread out.

This interpreter implements bfold, bdistr, bfilter and bunion directly in the underlying (imperative) language. The bdistr primitive corresponds to scheduling for execution all function applications at once, i.e.:

\[
\text{bdistr } f \text{ bag}(e_1, e_2, \ldots, e_n) \Rightarrow \text{bag}((f \text{ e}_1), (f \text{ e}_2), \ldots (f \text{ e}_n))
\]

bfold is implemented through the construction of a binary tree of applications of the collating function \( f \). Although in this preliminary version the issue of communication overhead is left out, we claim that the scalability of this approach is not impaired by communication overhead, which should affect at least equally both the list approach and the bag one.

7.3 Test Programs

In order to analyze the performances of the bag we have developed two simple programs: one computing the cardinality of a generic collection of objects (containing 2048 elements) and the other the cartesian product of two generic collection of objects (containing 64 elements); two implementation are compared on an hypothetic multiprocessor system: one based on bags and the other on lists.

The test target problems which seem quite suited for a “data parallel” approach. They are designed to show both that bags are useful in a single processor context and that they can exploit the potentialities of parallel architectures.

7.3.1 Cardinality

The bag version is:

\[
card :: \text{bag } t \rightarrow \text{num}
\]

\[
\text{card } b = \text{bfold } (+) \text{ 0 (bdistr one } \text{ b)}
\]

where

\[
\text{one } x = 1
\]
The list version is:

\[
\text{card :: list } t \rightarrow \text{num}
\]
\[
\text{card } [] = 0
\]
\[
\text{card } a : l = 1 + (\text{card } l)
\]

### 7.3.2 Cartesian Product

The \textit{bag} version is:

\[
\text{cartprod :: bag } t \rightarrow \text{bag } q \rightarrow \text{bag } (t, q)
\]
\[
\text{cartprod } b1 \ b2 = \text{bfold} \ \text{bunion} \\
\quad \quad \quad \quad \quad \quad \text{emptybag} \ (\text{bdistr} \ (cp \ b1) \ b2)
\]
\[
\text{where}
\]
\[
\text{cp } b \ x = \text{bdistr} \ g \ b
\]
\[
\text{where}
\]
\[
\text{g } a = (a, x)
\]

The list version is:

\[
\text{cartprod :: list } t \rightarrow \text{list } q \rightarrow \text{list } (t, q)
\]
\[
\text{cartprod } [] \ 12 = []
\]
\[
\text{cartprod } (a : l1) \ 12 = (\text{parlist} \ a \ 12) ++ \\
\quad \quad \quad \quad \quad \quad \text{cartprod} \ l1 \ 12)
\]
\[
\text{where}
\]
\[
\text{parlist} \ x \ [] = []
\]
\[
\text{parlist} \ x \ (a : l) = (x, a) : (\text{parlist} \ x \ l)
\]

Figure 4: Speed up for the cartesian product benchmark.

Figure 5: Scalability of the results: maximum of the speed ups.

### 8 Discussion

According to the tests we have performed the \textit{bag} data structure gives much better figures than the list on distributed systems both in terms of execution time and speed up.

Concerning the execution time (figures 1 and 2), we are impressed by two facts: the first is that already in a single processor framework \textit{bags} can perform much better than lists: as we already observed, this is due to the fact that there are certain classes of problems that are more tailored for a \textit{"bag"-approach} rather than a \textit{"list"-approach"}. The second is that the curve for \textit{bags} is steadily decreasing in the range examined, while the one for the list is constant, meaning that the degree of parallelism obtained with bags is higher than the one with lists.

Regarding the speed up, we observe that the \textit{bag} data structure overperforms the list one of order of magnitude (figures 3 and 4): the reason is quite obvious since the \textit{bag} is an intrinsic parallel data structure, as we have previously described. It seems also significant that both the tests gives quite the same outputs in terms of speed up for the \textit{bag} showing an interesting constance.

However what we think is the most remarkable feature is that the \textit{bag} implementation offers much space
of improvement in terms of scalability of the results. This can be easily seen analyzing figure 5: here we have fixed the number of processors and we analyze how the maximum speed up varies as a function of the size of the bag: we notice that this function is strictly increasing with a good slope and this lead us to think that the same situation will also happen for bigger sizes; this means both that this strategy allows to fully exploit the intrinsic parallelism of the problem and that, as long as we do not exceed the maximum degree of parallelism, adding processors results in bettering the the performances. Hence we can conclude that our system shows scalable performances.

9 Conclusions and Open Questions

In this paper we have described a different approach to the parallel execution of functional languages, the data parallel one. We have introduced a collection particularly suited for this approach, the bag, and we have given benchmarks on how well it works: we have demonstrated that it is very well suited for distributed systems since not only does it enhance the degree of parallelism but also it shows an interesting scalability of the results. Finally, we would like to stress the fact that the bag is not supposed to be a mutual exclusive alternative to the list, since it is trivial that for certain applications list are much better suited.

There are a lot of issues that are left open by our research, and perhaps the hardest one is about infinite bags. The problem in this framework is that no order over the element of a bag is imposed, hence a "lazy" approach consuming one element after the other would be highly unsafe, since the element we are interested in could be found only after an infinite search.

It would also be interesting to study how it is possible to couple this data parallel approach with the standard process parallel one.

Acknowledgments

We thanks the anonymous referees for their detailed and precious comments.

References


