STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLIX**, Number 1, 2004

UNBOUNDED AND BOUNDED PARALLELISM IN BMF. CASE-STUDY: RANK SORTING

VIRGINIA NICULESCU

Abstract. BMF is a formalism that allows us to design parallel programs independently of the target architecture, and to transform the programs into more efficient programs using equational reasoning. We show in this paper that even the abstractness of BMF is very high, bounded and unbounded parallelism can be expressed in this model, and also that BMF allows us to transform a program into different variants, each of them being more appropriate for a specific architecture type. We consider the case-study of rank sorting, for which we construct first a general unbounded parallel program. Then, we transform the program for bounded parallelism, by imposing a limited number of processors. Three variants are obtained. The implementations of the programs onto shared memory, distributed memory, and pipeline architectures are analyzed, too.

1. Introduction

The problem of constructing parallel software is much more difficult than for the sequential case. This is ultimately because there is no longer a single computation model to play the role that the von Neumann model plays in sequential computing. The goals of parallel software construction are also more ambitious. To be useful over any significant period of time, a parallel program must be able to be executed by many parallel computers, differing internally in significant ways. Thus the programmer's problem is not only to build an optimal program for a particular parallel computer, but to build one that is optimal for many different parallel computers [6].

2. Bird-Meertens Formalism

Bird-Meertens formalism (BMF) on lists was originally created for the design of sequential programs [1], but it becames very popular in the parallel setting. In BMF, higher-order functions (functionals) capture, in an architecture-independent

Received by the editors: September 2004.

2000 *Mathematics Subject Classification.* code, code.

1998 *CR Categories and Descriptors.* code [**Topic**]: Subtopic – *Detail*; code [**Topic**]: Subtopic – *Detail* .

way, general idioms of parallel programming, which can be composed for representing algorithms. BMF functionals use elementary operators and functions as parameters, so that a BMF expression represents a class of programs which can be reasoned about, either taking into account particular properties of the customizing functions or independently of them [5, 2, 3].

2.1. **BMF Notation.** The basic data structure it is considered the non-empty list: $[\alpha]$, where α is the elements type. Function application is denoted by juxtaposition, considering the tightest binding and association to the left.

The simplest functional of BMF is *map*, which applies a unary function f, defined on the elements type, to each element of a list:

$$map\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n] \tag{1}$$

The functional *map* is highly parallel since the computation of f on different elements of the list can be done independently if enough processors are available. The elements of a list can be lists again and f may be quite a complex function or composition of functions.

In addition to the parallelism of *map*, BMF allows to describe tree-like parallelism. This is expressed by the functional *red* (for reduction) with a binary associative operator $\oplus$:

$$red\ (\oplus)\ [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n \tag{2}$$

Reduction can be computed on a binary tree with $\oplus$ in the nodes. The time of parallel computation depends on the depth of the tree, which is $\log n$ for an argument list of length n.

Other functionals may be defined, but these two are the most important and the most used.

Functions are composed in BMF by means of functional composition $\circ$, such that $(f \circ g)\ x = f\ (g\ x)$. Functional composition is associative and represents the sequential execution order.

BMF expressions - and, therefore, also the programs specified by them - can be manipulated by applying semantically sound rules of the formalism. Thus, we can employ BMF for formally reasoning about parallel programs in the design process. BMF is a framework that facilitates transformations of programs into each others. A simple example of BMF transformation is the map fusion law:

$$map\ (f \circ g) = map\ f \circ map\ g \tag{3}$$

If the sequential composition of two parallel steps on the right-hand side of Eq. (3) is implemented via a synchronization barrier, which is often the case, then the left-hand side is more efficient and should be preferred.

A very important skeleton in BMF is the homomorphism that reflects the divide-and-conquer algorithms [2, 3]. Still, we will not refer to it in this paper.

We are going to focus on how BMF expresses unbounded and bounded parallelism.

3. Bounded and Unbounded Parallelism

For serial algorithms, the time complexity is expressed as a function of n – the problem size. The time complexity of a parallel algorithm depends on the type of computational model being used as well as on the number of available processors. Therefore, when giving the time complexity of a parallel algorithm it is important to give the maximum number of processors used by the algorithm as a function of the problem size. This is referred to as the algorithm's *processor complexity*. For example, a serial algorithm to find the maximum of a set with n elements has complexity $O(n)$, since it requires $n-1$ comparisons. In contrast, a trivial parallel algorithm for the same problem has time and processor complexities $O(\log n)$ and $O(n)$ respectively.

The synthesis and analysis of a parallel algorithm can be carried out under the assumption that the computational model consists of p processors only, where $p \geq 1$ is a fixed integer. This is referred to as *bounded parallelism*. In contrast, *unbounded parallelism* refers to the situation in which it is assumed that we have at our disposal an unlimited number of processors. Let us assume that a parallel algorithm A solves a problem of size n on p processors. If there exists a polynomial F such that for all n, $p < F(n)$, then the number of processors is said to be polynomially bounded; otherwise it is polynomially unbounded.

From a practical point of view algorithms for bounded parallelism are preferable. It is more realistic to assume that the number of processors available is limited. Although parallel algorithms for unbounded parallelism, in general, use a polynomially bounded number of processors(e.g. $O(n^2)$, $O(n^3)$, etc.) it may be that for very large problem sizes the processors requirement may become impractically large. However, algorithms for unbounded parallelism are of great theoretical interests, since they give limits for parallel computation and provide a deeper understanding of a problem's intrinsic complexity.

So, because of these, the right way for designing parallel programs is to start from unbounded parallelism and then transform the algorithm for bounded parallelism. There are two methods for carrying out such transformations. One is the decomposition of the problem into subproblems of smaller sizes, and another is the decomposition of the algorithm into substeps in such a way that each of them can be executed using a smaller number of processors.

The BMF programs usually reflect the unbounded parallelism. The functional *map*, for example, has the time complexity equal to $O(1)$ with a processor complexity n (n is the list's length).

In order to transform BMF programs for bounded parallelism, we may introduce the type $[\alpha]_p$ of lists of length p, and affix functions defined on such lists with the subscript p, e.g. map_p [3]. Partitioning of an arbitrary list into p sublists, called *blocks* may be done by the *distribution* function:

$$dist^{(p)} : [\alpha] \to [[\alpha]]_p \tag{4}$$

The following obvious equality relates distribution with its inverse, flattening: $red(|) \circ dist^{(p)} = id$, where $|$ means concatenation.

Using these, we obtain the following equality for the functional *map*:

$$map\ f\ = flat \circ map_p\ (map\ f) \circ dist^{(p)} \tag{5}$$

where the function $flat : [[\alpha]]_p \to [\alpha]$ transforms a list of sublists into a list by using concatenation ($flat = red(|)$).

For reduction, the transformation for bounded parallelism is as it follows:

$$red(\oplus)\ = red_p(\oplus) \circ map_p\ (red(\oplus)) \circ dist^{(p)} \tag{6}$$

We will consider that only the functions with subscription p will be distributed over the processors. The others will be sequentially computed.

4. Case-study: Rank Sort

The idea of the rank sort algorithm is the following: determine the rank of each element in the unsorted sequence and place it in the position according to its rank. The rank of an element is equal to the number of elements smaller than it [4].

Rank sort is not exactly a good sequential sorting algorithm because the time complexity in the sequential case is: $O(n^2)$ (n is the length of the sequence). But this algorithm leads to good parallel algorithms.

Using this simple example, we are going to illustrate that the abstract design using BMF can comprise different cases that may appear at the implementation phase: shared memory versus distributed memory and pipeline computation, and different numbers of processors. So, the abstractness of this formalism does not exclude performance.

4.1. **BMF design.**

By using the definition of the method we arrive to the following simple BMF program:

$$\begin{aligned}
&rank : [\alpha] \to [\alpha] \\
&rank\ l = map\ (count\ l)\ l \\
&\\
&count : [\alpha] \times \alpha \to [\alpha] \\
&count\ l\ x = red(+) \circ map\ (f\ x)\ l \\
&\\
&f : \alpha \times \alpha \to \alpha \\
&f\ x\ y = \begin{cases} 1, \text{ if } x \geq y \\ 0, \text{ if } x < y \end{cases}
\end{aligned} \tag{7}$$

A simple and obvious transformation can be made:

$$red(+) \circ map\ (f\ x)\ l = red(+)\ (map\ (f\ x)\ l) \tag{8}$$

which simple means that the application of the function $f\ x$ is made in the same step with the reduction, not in a separate sequent step.

4.2. **Unbounded parallelism.** We will analyze first the unbounded parallelism. For computing the first functional *map* we need a number of processors equal to the length of the input list – n. Each application of the function *count* l is a reduction which can be computed with $O(\log n)$ time complexity and $O(n)$ processor complexity. So, for whole program the unbounded time complexity is $O(\log n)$ with the processor complexity equal to $O(n^2)$.

4.3. **Bounded parallelism.** For bounded parallelism we need to transform the program by imposing the number of processors to be equal to p. For the transformation we use the function $dist^{(p)}$.

As it may be noticed from the specification, the algorithm contains two phases: one represented by the function *map* and the other represented by the function *count*, each of them having the list l as argument. The function $dist^{(p)}$ simple divided the argument list into p balanced sublists. So we may apply it for the computation of the function *map*, or for the computation of the function *count*, or for both.

There are two cases that we have to take into account:

(1) $p \leq n$,
(2) $n < p \leq n^2$

4.3.1. *Case* $p \leq n$. If the number of processes is less or equal to the size of the sequence, than the function $dist^{(p)}$ may be applied only once: for the function *map*, or for the function *count*.

In the first case we obtain the following BMF program, by using the equational rule (5):

$$\begin{array}{l} rank\ l = (flat \circ map_p\ (map\ count\ l) \circ dist^{(p)})\ l \\ count\ l\ x = red(+)\ (map\ (f\ x)\ l) \end{array} \tag{9}$$

This means that each processor sequentially computes the ranks for n/p elements.

If we applied the function *dist* to the function *count* we obtain the following BMF program:

$$\begin{array}{l} rank\ l = map\ (count\ l)\ l \\ count\ l\ x = red_p(+) \circ map_p\ (red(+)) \circ (map\ (f\ x)) \circ dist^{(p)}\ l \end{array} \tag{10}$$

Here we have used the equational rule (6). This program sequentially computes the ranks of the elements, but the the ranks are computed in parallel using p processors. For computing the rank of an element x, the processors compare x

```
forall (i = 0; i < p; i++) do in parallel
  for (k = 0; k < n/p; k++) do
    global_read(A[i * n/p + k], ak);
    r = 0;
    for (j = 0; j < n; j++) do { count the numbers less than ak}
      global_read(A[j], aj);
      if (aj < ak)  r++;  fi
    rof
    global_write(r, R[i * n/p + k]);
  rof
llarof
```

FIGURE 1. The SM program for $p \leq n$

with all their n/p local elements and compute local ranks; then the local ranks are added.

The two cases reflect different ways for algorithm decomposition in substeps, which can be computed with p processors. We consider that the functions with the subscript p will be computed in parallel on the p processors, and the functions without subscription will be computed sequentially. This leads to the following time complexities: (n^2/p) for the first case, and $(n^2/p + n \log p)$ for the second. Obvious the first is the best.

Implementations for $p \leq n$. On *Shared Memory* (SM) architectures the list l is shared by all processors, so normally we will choose the first case with the better time complexity. We may transform the correspondent BMF program into the PRAM-like program described in the Figure 4.3.1.

Each processor makes n^2/p readings and n/p writings from/in the shared memory. If we consider a CREW architecture the complexity is $(n^2/p+\alpha(n^2/p+n/p))$, where α is the unit time for shared memory access.

On a *Distributed Memory* (DM) architecture, the second alternative is better since we have the list distributed over the processors. An element is broadcasted to all the processors during the step that computes its rank - so there are n steps. Each processor compares the current received element with the local elements and computes a local rank. Local ranks are summed using a tree like computation that represents the reduction. A psudo-MPI program is described in the Figure 4.3.1.

The time complexity is given by the following expression: $n(n/p+\log p(1+\beta)+ b\beta)$, where β is the unit time for communication, and constant b reflects the time for broadcast.

Pipeline architectures may also be used for implementing this algorithm. If there are n processors, each processor has a local value $a[i]$. The elements of

```
Rank(mypid) :
  for (i = 0; i < n; i++) do
    Bcast(A[i]); { the root is the process that contains A[i] }
    rl = ComputeLocalRank(A[i]);
    Reduce(rl, mypid);
  rof
```

FIGURE 2. The DM program for $p \leq n$

the list are piped into the pipeline, and the rank is updated in each processor by comparing the piped value with the local value. The current rank is also piped into the pipeline. If $p < n$ than each processor makes more comparisons at each step. The time complexity, for this implementation, is: $(n+p-1)(n/p+\beta)$, where β is the unit time for communication between two processors.

4.3.2. *Case* $n < p \leq n^2$. If we have more than n processors and $p = q * r$, we may use the function *dist* for *map* and also for *count* computations. So, we arrive to the following BMF program:

$$\begin{array}{l} rank\ l = (flat \circ map_q\ (map\ count\ l) \circ dist^{(q)})\ l \\ count\ l\ x = red_r(+) \circ map_r\ (red(+)) \circ (map\ (f\ x)) \circ dist^{(r)}\ l \end{array} \tag{11}$$

The time complexity for this case is $(n/q)(n/r + \log r)$.

On a SM architecture the program differs by the program presented in Figure 4.3.1 with the fact that for each element the function *count* is computed using a tree-like computation. Each processor makes $n^2/p + (n/q)\log r$ readings and $(n/q)\log r$ writings from/in the shared memory.

For a DM architecture the processes may be arranged on a $q \times r$ mesh, and the data distribution of the first line may be replicated on the other lines of processors. Each line computes the ranks of a sublist with n/q elements. The time complexity is $n^2/p + n/q(\log r(1+\beta) + b\beta)$.

A pipeline architecture with more than n processors is no useful for this problem.

5. CONCLUSIONS

We have analyzed here the developing of parallel programs for rank sorting using Bird-Meertens formalism. First, a general unbounded parallel program is constructed. Then, we transform the program, by imposing a limited number of processors. Three variants are obtained, two for the case $p \leq n$ and one for the case $n < p \leq n^2$. Then the implementations of the programs onto shared memory (SM), distributed memory (DM), and pipeline architectures are analyzed. If $p \leq n$ the first BMF variant is used for implementation on SM architectures; for DM and also for pipeline architectures the second variant is most appropriate. A

pipeline architecture with more than n processors for this problem is not useful. If $n < p \leq n^2$ the implementations on SM and DM architectures start from the same BMF program. For the DM case data replication is going to be used.

Time complexities for each case are analyzed too.

Parallel programs for rank sorting have been developed before, rank sorting algorithm being one that shows that a poor sequential algorithm for a problem may lead to very good parallel algorithms. It has been considered especially for sorting on SM machines [7]. We have shown here in a formalized way that we may successful use it for DM and pipeline architectures. Also, we have formally analyzed the case when the number of processes p is between n and n^2.

Yet, with this example, a more important thing has been emphasized: BMF programs can be formally transformed for bounded parallelism and the variants which are obtained may be analyzed later for choosing the right one for a specific target machine.

The abstract design using BMF can comprise different cases that may appear at the implementation phase: shared memory versus distributed memory and pipeline computation, and different numbers of processors. So, the abstractness of this formalism does not exclude performance.

References

[1] R. Bird. *Lectures on Constructive Functional Programming*. In M. Broy editor, Constructive Methods in Computing Science, NATO ASI Series F: Computer and Systems Sciences, Vol. 55, pg. 151-216. Springer-Verlag, 1988.

[2] M. Cole. *Parallel Programming with List Homomorphisms*. Parallel Processing Letters, 5(2):191-204, 1994.

[3] Gorlatch, S., *Abstraction and Performance in the Design of Parallel Programs*, CMPP'98 First International Workshop on Constructive Methods for Parallel Programming, 1998.

[4] Knuth, D.E., *The Art of Computer Programming. Vol. 3 Sorting and Searching*, Addison-Wesley, 1973.

[5] D.B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[6] Skillikorn, D., *Architectures, Costs, and Transformations*, CMPP'98 First International Workshop on Constructive Methods for Parallel Programming, 1998.

[7] Wilkinson, B., Allen, M., *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall, 2002.

Department of Computer Science, Babeş-Bolyai University, Cluj-Napoca
E-mail address: vniculescu@cs.ubbcluj.ro