7M−6 LU Decomposition with Arrays in Miranda

Martin Santavy - 讃多美 マーティン
九州大学工学部情報工学科

Introduction

We describe a small set of simple, yet powerful operations that allow us to manipulate large segments of data. The operations are designed to utilize the power of parallel environments. At the same time, they are flexible enough to be useful theoretical tools that can be part of the design of parallel algorithms and reasoning about them.

In parallel environments, loops with indices and references to individual data items have to be "parallelized", which can often be a difficult, if not unsolvable, problem. In our approach, we want to avoid it altogether by manipulating arrays in their entirety instead of accessing their individual elements one by one.

Also, instead of trying to modify the structure of a programming language to allow the programmer to execute parts of her program in parallel, we parallelize the data and leave the language to manipulate them sequential. The efficiency of the programs will lie in the efficiency of the basic tools, "primitives", that the programmer uses to manipulate the data. These tools must be designed in a way that permits an efficient implementation in most parallel environments.

Miranda is a polymorphic, strongly-typed, functional language with lazy evaluation, abstract data types, and currying. We use its power to combine our primitives in a simple and elegant way. In the design of the primitives themselves, however, we settle for only a theoretical consideration of their possible parallel implementations and implement them in Miranda in an inefficient, sequential, but demonstrative way.

Definition

An array is a multidimensional, rectangular structure that can be fully described by its shape and values. To describe the shape, we use a finite vector of natural numbers. The numbers indicate the size of the array along the corresponding dimensions; the length of the shape vector is the dimensionality of the array. The values of the array form a vector of length equal to the product of the shape. All values are of the same type.

Alternatively, an array can be viewed as a rectangular grid with a unique value associated with each gridpoint or as a list of lists, with each sublist representing a subarray of the array.

Representation

When a shared-memory machine is used, the array can be stored in a continuous block of memory in the row-major order. Processors can then independently manipulate different parts of the array. When a machine with no shared memory is used, the array must be distributed among local memories of individual processors. This can be done either by using the combination of the processor address and the local offset to store the array in the row-major order or by using some communication scheme among processors, e.g. a tree or a mesh.

In Miranda, we use an array constructor, `A`, to construct a pair of a shape vector and a value vector, both of type `[num]`. The type of the array is then called `reparray`. Its synonym, `array`, is used as a name of the abstract data type that includes the array representation, `reparray`, and a set of basic function to manipulate it, `first, rest, cat, isempty`, etc.:

```
reparray ::= A [num] [num]
array == reparray
absttype array
with
  first, rest :: array->array
  cat         :: array->array->array
  isempty     :: array->bool
  ....
```

A new array can be created using function `arr ss vs`, which returns an array of shape `ss` with values taken from vector `vs`. If necessary, vector `vs` is repeated.

Basic Operations-Primitives

The basic set of operations must include functions that give information about the structure of the array. Examples of structural-information functions are the function `shp x` that returns a list of values representing the shape of the vector `x` and the function `issingle x` that returns the boolean value `True` if and only if the array `x`, when viewed as a list of subarrays, has only one element.

```
shp (A ss vs) = ss
issingle (A (s:ss) vs) = s=1
```

Functions `first`, `rest`, `cat` return the first subarray of an array, everything but the first subarray of an array, and a catenation of two arrays along the 0-th dimension, respectively. The functions follow a simple set of rules. An example of a such a rule is

```
cat (first x) (rest x) == x
```

which holds for for any x.

Functions `tk` and `dp` treat arrays as lists of lists and behave similarly to Miranda's functions `take` and `drop`. In addition to these two functions, we define function `bk` that returns a specified number of subarrays, starting from a specified position. Since arrays are multidimensional, the functions accept a list of values indicating the number of elements to be taken or dropped in each dimension.

```
tk a == bk a []
dp a == bk [] a
```

Using these primitives, we can construct other functions, e.g. a function `cnst c x` that returns the array `x` with all values replaced by `c`:

```
cnst n x = arr (shp x) (repeat n)
```

LU Decomposition with Arrays in Miranda
Martin Santavy
Department of Computer Science
and Communication Engineering
Kyushu University

Higher-Order Operations

A *higher-order operation* accepts functions as its arguments and produces a function as its result.

The higher-order operation `mu1 f x` applies function `f` to every subarray of array `x`, which is viewed as a list of its subarrays. Operator `mu1` is defined as:

```
mu1 f x
= arr (0:shp (f (first x))) [], hd (shp x)=0
= cat (f (first x))(mu1 f (rest x)),otherwise
```

The operator `mu1'` is a simple generalization of `mu1`. It accepts an additional numerical parameter that determines along which dimension the function-argument should be applied.

```
mu1'(n+1) f  = mu1 (mu1' n f)
mu1'  0   f  = f
```

The higher-order operations `op1` and `op2` extend common unary and binary operations to arrays. When the sizes of the arrays do not match, `op2` tries to replicate the "smaller" argument to match the size of the other argument.

```
op1 f (A ss vs)
= A ss (map f vs)

op2 f (A s1 v1) (A s2 v2)
= A s1 (map2 f v1 v2), s1=s2
= mu1 (op2 f (A s1 v1)) (A s2 v2), #s1<#s2
= mu1 (op2 g (A s2 v2)) (A s1 v1), #s1>#s2
  where g a b = f b a
```

The higher-order operation `upd a b f` (update) restricts the function `f` to the part selected by `bk a b`, while leaving the rest of the array intact.

The `it` (iterate) operator applies a given function recursively on smaller and smaller parts of a given array. It is not a primitive.

```
it ss1 ss2 f x
= x, isempty x
= upd ss1 ss2 (it ss1 ss2 f) (f x), otherwise
```

Example: LU Decomposition Algorithm

Let us consider the following Pascal routine to decompose matrix `A` into a product `L*U` of a lower-diagonal matrix `L` and an upper-diagonal matrix `U`. The diagonal of `L` is formed of ones. At the end of the routine, the lower-diagonal part of matrix `A` is replaced by the lower-diagonal part of matrix `L` (excluding the diagonal) and the upper-diagonal part of matrix `A` is replaced by the upper-diagonal part of matrix `U` (including the diagonal).

```
procedure LU {n:integer; var A:matrix)
var i,j,k:integer;
begin for k=0 to n-1 do
      begin for i:=k+1 to n-1 do {step 1}
                a[i,k]:=a[i,k]/a[k,k];
            for i:=k+1 to n-1 do {step 2}
               for j:=k+1 to n-1 do
                  a[i,j]:=a[i,j]-a[i,k]*a[k,j];
      end
end;
```

It is not difficult to verify that this algorithm indeed finds an LU decomposition of a given matrix. Let us simply ignore the problem of division by zero when `a[kk]=0` here.

Tho loops, however, do not describe the real idea of the algorithm. The real idea is that we subsequently take smaller and smaller submatrices of matrix A, and for each submatrix we apply two steps. Firstly, we divide its first column (except for the upper-left-corner element) by the upper-left-corner element. Secondly, we subtract the outer product of this column with the first row (without the upper-left-corner element) from the lower right principal minor of the submatrix (i.e. what is left after removing the first row and column).

In our system, the upper-left-corner element of matrix `x` is accessed as `first(first x)`, while its first column (without the first element) is `bk [n-1,1] [1,0] x where n=hd(shp x)`. Therefore, step 1 can be written as:

```
step1 x
= upd [n-1,1] [1,1]($divd (first(first x)))x
  where n = hd(shp x)
```

Operation `divd` is an array extension of the regular divide operation.

The outer-product, needed in step 2, is very easy to define when the operator `mu1'` is used.

```
op x y = mu1' k (mult x) y where k=dim y
```

where `mult` is defined, predictably, as `op2(*)`. The first column and the first row (both without their first element) of a matrix `x` can be accessed by `mu1 first(rest x))` and `rest(first x)`, respectively. The lower right principal minor is simply `dp [1,1]`, or `bk [] [1,1]`. Therefore, step 2 of the algorithm can be written as

```
step2 x
= upd [] [1,1] ($sub
  (op(mu1 first(rest x)) (rest(first x)))) x
```

The consecutive applications of steps 1 and 2 are simply

```
lu = it [] [1,1] (step2.step1)
```

Conclusion

The Pascal routine in the example contains a triply-nested loop. Our Miranda equivalent contains none. We see two main positive sides of our approach:

- when the primitives are implemented in parallel, the whole algorithm is inherently parallel, and
- when the algorithm is free of unnecessary loops and indices, it is much easier to see its basic ideas and reason about it.

Acknowledgements

I thank Lenore Mullin and Nathan Freedman for their support and encouragement.

References

[1] J. Backus, Can Programming be liberated from the von Neumann style: A functional style and its algebra of programs, Communications of the ACM 22, no.8, pp. 613-641, Aug. 1978.

[2] J. Bird, P. Wadler, Introduction to Functional Programming, Prentice Hall, 1988

[3] L.M.R. Mullin, A Mathematics of Arrays, Ph.D. thesis, Syracuse University, 1988.

[4] L.M.R. Mullin, G. Gao, M. Santavy, & B. Tiffou, Formal Program Derivation for a Shared-Memory Architecture: LU-Decomposition, McGill Univ., TR in progress, May 1990.

[5] M.Santavy, Arrays in Miranda, IEICE Technical Report, vol.90, no.101, pp. 19-26, Dec. 1990.

[6] D.A. Turner, Miranda: a non-strict functional language with polymorphic types, in J.-P. Jouannaud, editor, Functional Programming Languages and Computer Architecture, Springer-Verlag, 1985.

[7] D.A. Turner, An overview of Miranda, SIGPLAN Notices, December 1986.