Array Algorithms

John E. Howland
Department of Computer Science
Trinity University
One Trinity Place
San Antonio, Texas 78212-7200
Voice: (210) 999-7364
Fax: (210) 999-7477
E-mail: jhowland@Trinity.Edu
Web: http://www.cs.trinity.edu/~jhowland/

January 27, 2005

Abstract

Array Algorithms are defined as functional algorithms where each step of the algorithm results in a function being applied to an array, producing an array result. Array algorithms are compared with non-array algorithms. A brief rationale for teaching array algorithms is given together with an example which shows that array algorithms sometimes lead to surprising results. [1]

Subject Areas: Array Algorithms, Computer Science Education, Computer Science Curriculum.
Keywords: Array Algorithms.

1 Introduction

In this paper, the term *array algorithm* is used in a context which goes beyond algorithms which use array data structures. Array algorithms use arrays or lists as their principle data structure and consist of operations which are applied to arrays producing arrays as results.

Array algorithms, because they deal with data in aggregate, involve different problem solving processes and require programming languages which easily support operations on arrays.

Array algorithms have been widely discussed in the literature. A search of the ACM digital library for *array algorithm* finds hundreds of citations [7, 9, 1] ... which deal with array algorithms. Almost all of these papers focus primarily on algorithms which use array data structures rather than the sense in which we have defined array algorithms. Metzger, Eisenberg and Peelle [6, 3, 4] have written on this subject, however their work was directed toward a specific programming language (APL).

Array algorithms provide a different perspective on problem solving which often leads to a different insight about the problem being solved. For this reason, it is important to include a treatment of array algorithms and array languages in the undergraduate curriculum. Computing Curricula 2001 Computer Science [2] does not address array programming in the sense of our definition.

[1] This paper has been accepted by the referees for publication in the Journal of Computing Sciences in Colleges, Volume 20, Number 4, April 2005. Copyright ©2005 by John E. Howland, all rights reserved.

However, because of changes in the design of processors which are now beginning to be used and will be standard in most systems in the future, it is necessary to consider program design techniques which will make efficient use of new processor technology. Recent trends in processor design are following a new direction in order to continue to follow Moore's Law (that computer processing power approximately doubles every 18 months). That new direction is processor designs from companies such as IBM, Intel, AMD, and others, have multiple processors on a single chip core rather than having designs of a single processor whose speed doubles about every 18 months. Recent multicore chip designs have individual processors which are actually slower than current generation single processor designs. This means that in order to achieve faster computing on such processors, processing must routinely be done in parallel. Array algorithms have the potential of being compiled to run in parallel for such processors.

2 Array Algorithms

In simple examples, the differences between array algorithms and non-array algorithms are subtle. For example, consider the algorithm which computes the average of a list of numbers. In C, this program `ave.c` might be written as:

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{ int sum, count, n;
  count = 0;
  sum = 0;
  while (1 == scanf("%d\n", &n))
   { sum = sum + n;
     count++;
   }
  printf("%f", (float)sum / (float)count);
  exit(0);
}
```

To run this program (after compiling) one could write

```
$ echo "1 2 3" | ave
2.000000
```

An array program, written in the J programming language, a functional array language, is expressed as:

```
$ echo "(+/ % #) 1 2 3" | jconsole
   (+/ % #) 1 2 3
2
```

The C average program deals with elements of an array or list, without explicitly storing them in an array, on an item by item basis, accumulating the sum and count. After processing all elements in the array, the average is formed by dividing the sum by the count. The J average program applies two functions (`+/` "sum") and (`#` "tally") to the entire array and then computes the average by dividing (`%` "divide"). The J program uses a functional composition rule `(f g h) x = (f x) g (h x)` to accomplish this task.

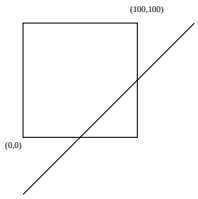

Figure 1: Clip a square to a line

3 Languages Which Support Array Algorithms

APL, J, Lisp, Scheme, and array classes for C++ and Java are examples of languages which have the potential for expressing array algorithms.

An array language should possess the following features for adequate expression of array algorithms:

- create arrays of any type, rank, and size
- function application on arrays producing array results
- functions should be "first class" data, i.e. we should be able to have arrays of functions and apply such function arrays
- higher level functions i.e. apply functions to functions producing functions as results

The J language [5] is used in this paper because of its simple syntax and concise expression of array operations. Any of the above mentioned languages could be used, making the paper substantially longer.

4 Polygon Clipping

To illustrate a non-trivial array algorithm, consider the well known algorithm for polygon clipping by Sutherland and Hodgman [8]. Polygon clipping reduces a polygonal surface extending beyond the boundary of some three-dimensional viewing volume to a surface which does not extend beyond the boundary. The Sutherland Hodgman algorithm is recursive, processing one line segment at a time considering each endpoint.

We restrict ourselves to the two-dimensional case of clipping a closed polygonal figure to a line. The array algorithm applies to three-dimensional data without changes to the J program.

Suppose we have the square:

```
   [ square =: 5 2 $ 0 0 100 0 100 100 0 100
  0   0
100   0
100 100
  0 100
```

and the line (in homogeneous form) as illustrated in Figure 1.

```
   [ line =: _1 1 50
_1 1 50
```

The array algorithm for polygon clipping (expressed in J) is:

```
pclip =: 4 : 0
a =. (( {. $ r =. y.) , 2) $ 1 0
pic =. (r ,. 1) +/ . * x.
a =. 2 (<"1 (i =. (-. (* pic) = * 1 |. pic) # i. {. $ y.) ,. 1) } a
q =. |: ((_1 + $ x.) , $ pic) $ pic
q =. (((i { r) * i { 1 |. q) - (i { 1 |. r) * (i { q)) % (i { 1 |. q) - i { q
r =. (pic > 0) # r
a =. 0 (<"1 ((0 >: pic) # i. $ pic) ,. 0) } a
a =. (-. 0 = a) # a =. , a
r =. (/: /: a) { r , q
r , (-. (1 {. r) -: _1 {. r) # 1 {. r
)
```

Applying `pclip` we have:

```
   line pclip 0 1 2 3 0 { square
  0   0
 50   0
100  50
100 100
  0 100
  0   0
```

which is illustrated in Figure 2. We can clip to the other side of the line by:

```
   (-line) pclip 0 1 2 3 0 { square
 50  0
100  0
100 50
 50  0
```

The J expression of this algorithm is remarkably short, due in part to the expressive power of J, but also to the array nature of the algorithm where loops and iterations are contained within the functions which are applied to arrays. This is a recurring feature of array algorithms. We explain the results formed in each of the ten lines of the program.

The first expression produces a table, named `a`, which will be used to code which side of the line/plane the points lie. A copy of the original data is also named (locally) `r` for later use. Initially, the algorithm assumes that column 1 of `a` codes all of the points as being on the correct side of the clipping boundary with a value of 1. Later, points which are clipped will be coded with the value 0 in column 1 and new intersection points which must be added will be coded in column 2 with a value of 2.

```
   a =. (( {. $ r =. y.) , 2) $ 1 0
1 0
1 0
1 0
1 0
1 0
```

The second line produces a vector of values which indicate which side of the line/plane (a positive value indicates the correct side of the line/plane) by computing a matrix product of the homogeneous representation of the points and the line/plane. This result is named (locally) `pic`.

```
   pic =. (r ,. 1) +/ . * x.
50 _50 50 150 50
```

The third line modifies column 2 of the table `a` to indicate that the one point, (100,0), which lies on the wrong side of the clipping line/plane must be replaced by two points along the clipping boundary which are determined by computing the intersection with the clipping boundary.

```
   a =. 2 (<"1 (i =. (-. (* pic) = * 1 |. pic) # i. {. $ y.) ,. 1) } a
1 2
1 2
1 0
1 0
1 0
```

The fourth line builds a table, named `q`, of two columns which consist of the vector `pic`.

```
   q =. |: ((_1 + $ x.) , $ pic) $ pic
 50  50
_50 _50
 50  50
150 150
 50  50
```

The fifth line computes a table, named `q`, of the intersection points with the clipping boundary of all lines/planes which go outside the clipping boundary.

```
   q =. (((i { r) * i { 1 |. q) - (i { 1 |. r) * (i { q)) % (i { 1 |. q) - i { q
 50  0
100 50
```

The sixth line computes a table, re-named `r`, of the points which are on the correct side of the clipping boundary.

```
   r =. (pic > 0) # r
  0   0
100 100
  0 100
  0   0
```

The seventh line modifies the table, `a`, to indicate, with a zero in column one, which points are clipped from the original data.

```
   a =. 0 (<"1 ((0 >: pic) # i. $ pic) ,. 0) } a
1 2
0 2
1 0
1 0
1 0
```

The eighth line produces a vector of the non-zero elements in `a` in row major order. This array encodes with the value 1 the respective elements of `r` and uses the value 2 for the respective elements of `q` which are the newly computed boundary intersection points. This vector is the mesh vector for properly ordering the points not clipped and the boundary intersection points.

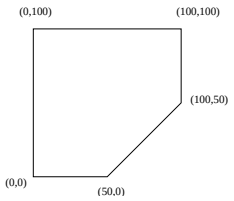

Figure 2: Square after clipping

```
   a =. (-. 0 = a) # a =. , a
1 2 2 1 1 1
```

The ninth line uses the grade-up function (`/:`) which produces a permutation of the indices of an array which would sort the array in ascending order. That applying grade-up twice produces the proper ordering of elements taken from a table formed by combining the tables `r` and `q` is a remarkable surprise. The ninth line re-arranges the elements in r and q by sorting a sort!

```
   (/: /: a) { r , q
  0   0
 50   0
100  50
100 100
  0 100
  0   0
```

This algorithm assumes that we start with a closed polygon. To ensure that the algorithm produces a closed polygon, the last line closes the polygon in the case where the first (and last) point is clipped.

```
   r , (-. (1 {. r) -: _1 {. r) # 1 {. r
  0   0
 50   0
100  50
100 100
  0 100
  0   0
```

5 Conclusions

Array algorithms involve a way of thinking about arrays of data and performing operations on the entire array. Array algorithms may be easily parallelized because the array item processing in each array operation is known to be independent from the item processing in any other array operation in the algorithm since the array operations are performed in sequence.

Also, the array thinking discipline leads to more general solutions which may be used to solve other problems by changing the functions being applied. For example, (using J) the matrix product of `a` and `b` is expressed as the J dot product `a +/ . * b`. Here we are talking about summing (`+/`) the products (`*`) of rows of `a` and columns of `b`. The dot product `a */ . = b` is useful for finding matches of rows of a and columns of b.

Teaching students to develop array algorithms gives them another way of looking at the problem solving process which sometimes gives new insight about the problem being solved as well as often producing an algorithm which may be easily parallelized. Parallel algorithms will be routinely required to achieve improved performance on new multicore processor designs. Array algorithms will play an important role in the development of parallel software.

References

[1] Bosse, Michael, "Real-world Problem-solving, Pedagogy, and Efficient Programming Algorithms in Computer Education", ACM SIGCSE Bulletin, 32(4):66-69, December 2000.

[2] *Computing Curricula 2001 Computer Science, Final Report*, The Joint Task Force on Computing Curricula, IEEE Computer Society and Association for Computing Machinery, IEEE Computer Society, December 2001.

[3] Eisenberg, Murray, and Peelle, Howard, "APL Thinking: Examples", ACM SIGAPL, Proceedings of the International Conference: APL in Transition, 17(4):433-440, January 1987.

[4] Eisenberg, Murray, and Peelle, Howard, "A Survey: APL Thinking", ACM SIGAPL Quote Quad, 21(2):5-8, December 1990.

[5] Hui, Roger K. W., Iverson, Kenneth E., *J Dictionary*, J Software, Toronto, Canada, May 2001.

[6] Metzger, Robert, "APL thinking finding array-oriented solutions", ACM SIGAPL, Proceedings of the International Conference on APL, 12(1):212-218, September 1981.

[7] Stallmann, Matthias F. M., "A One-way Array Algorithm for Matroid Scheduling", Proceedings of the third annual ACM symposium on Parallel Algorithms and Architectures, Pages: 349-356, Hilton Head, South Carolina, 1991.

[8] Sutherland, Ivan and Hodgman, Gary, "Re-entrant Polygon Clipping", Communications of the ACM, 17(1):32-42, January 1974.

[9] Zhao, Yihong, Deshpande, Prasad, Naughton, Jeffrey, "An Array-based Algorithm for Simultaneous Multidimensional Aggregates", Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, Pages: 159-170, Tucson, Arizona, 1997.