Array Algorithms

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

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.

Overview of Presentation

- Background of the Problem
- Definition of an Array Algorithm
- Several Examples
- Conclusions

Background

"... the times they are a-changin' ..." : Bob Dylan

- Moore's Law: (April 19, 1965 issue of *Electronics*) ... innovations in technology would allow a doubling of the number of transistors in a given space every year ...
- Moore updated this (1975) to a doubling of the number of transistors every two years to account for the growing complexity of chips
- We have experienced this exponential growth for 40 years

Background (continued)

- Intel Gives Up on Speed Milestone (Wall Street Journal, October 15, 2004)

  > The company said it no longer plans to offer its Pentium 4 chip for desktop computers at a clock speed of four gigahertz, a target that had alread slipped from the end of this year until sometime in 2005. The fastest member of that chip family is now 3.6 gigahertz.

- Intel is now focusing on increasing cache memory and putting multiple processors on a single chip.

- While Moore's law appears to continue to hold, the more dense chips will not include the previous benefit of allowing higher clock speeds due to heat dissapation and current leakage problems.

Background (continued)

- Intel, IBM and other processor manufacturers have announced they too will be following the strategy of providing multiple processor chips which are not necessarily faster than their current chips.
- New Chips Pose a Challenge to Software Makers (Wall Street Journal, April 14, 2005, B3)
- How will our programs go faster in the future?

Background (continued)

- Parallel programming will have to become the standard approach in all of the programs we write (if we wish to go faster).
- We need new programming languages.
- We need new programming methodoligies.

Array Algorithms

- 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 principal data structure and consist solely of functions which are applied to arrays producing arrays as results.

- Of course, occasionally, array algorithms have one element results or arguments.

Array Algorithms (continued)

Array algorithms:

- involve different problem solving processes
- require programming languages which support primitive operations on arrays or have libraries of array operations
- C++ or Java (with the inclusion of appropriate array operation classes and libraries)
- APL, Lisp, Scheme, Fortran90, MatLab, J

Array Thinking

- Array algorithms provide a different perspective on problem solving which often leads to a different insight about the problem being solved.

- Array algorithms require thinking about the data, and how it may be organized so that operations may be performed on all of the items in the array rather than what can be done with the elements of the array on an item by item basis.

- Array thinking helps formulate a problem solution so that operations may be performed on collections of data (in parallel) on appropriate hardware.

Example 1 (average)

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 shown on the next slide. This program reads the elements of an array or list and accumulates the sum of the elements as each number is read.

Example 1 (average continued)

```
#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);
}
```

Example 1 (average continued)

To run this program (after compiling) one could write

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

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.

Example 1 (average continued)

In the array programming examples which follow, the J programming language is used primarily for its concise presentation of array algorithms. Other languages, such as APL, Lisp, Scheme, Fortran90, Matlab, C++, or Java (the latter two with appropriate array operation classes or libraries) could have been used at the expense of brevity of presentation.

Example 1 (average continued)

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 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.

Example 2 (computing differences)

Given some data values,

$$x_i, i = 0, 1, \ldots, 10 : -125, -64, -27, -8, -1, 0, 1, 8, 27, 64, 125 \tag{1}$$

compute (iterative algorithm)

$$x_1 - x_0, x_2 - x_1, \ldots, x_{10} - x_9 \tag{2}$$

$$61, 37, 19, 7, 1, 1, 7, 19, 37, 61 \tag{3}$$

Example 2 (computing differences continued)

Given some data values,

```
   [x =: (_5 + i. 11) ^ 3
_125 _64 _27 _8 _1 0 1 8 27 64 125
```

compute (array algorithm)

```
   1 }. x
_64 _27 _8 _1 0 1 8 27 64 125
   _1 }. x
_125 _64 _27 _8 _1 0 1 8 27 64
   (1 }. x) - _1 }. x
61 37 19 7 1 1 7 19 37 61
```

Example 3 (computing line lengths)

Suppose we have a text array (file):

```
   [text =: 0 : 0
Now is the time
to do
the work and this
is the work that must be done.
)
Now is the time
to do
the work and this
is the work that must be done.
```

Example 3 (computing line lengths continued)

```
   nl =: 10 { a.  NB. a new-line character
   [x =: _1 , (text = nl) # i. # text
_1 15 21 39 70
   _1 + (1 }. x) - _1 }. x
15 5 17 30
   text
Now is the time
to do
the work and this
is the work that must be done.
```

Example 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.

Example 4 (polygon clipping continued)

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.

Example 4 (polygon clipping continued)

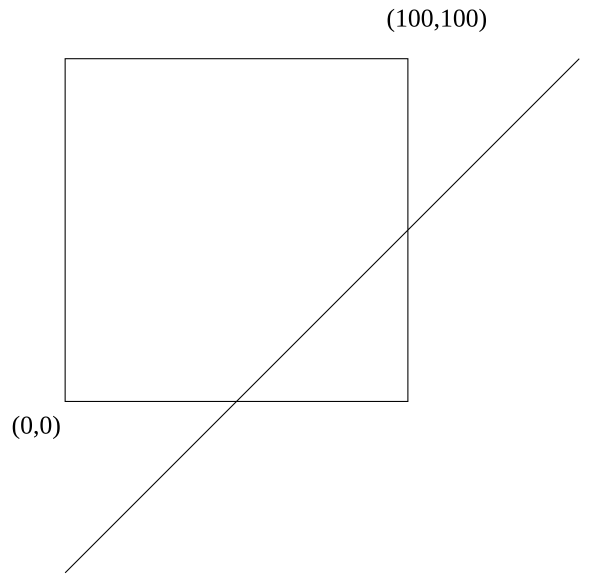

Figure 1: Clip a square to a line

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

Example 4 (polygon clipping continued)

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
)
```

Example 4 (polygon clipping continued)

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.

Example 4 (polygon clipping continued)

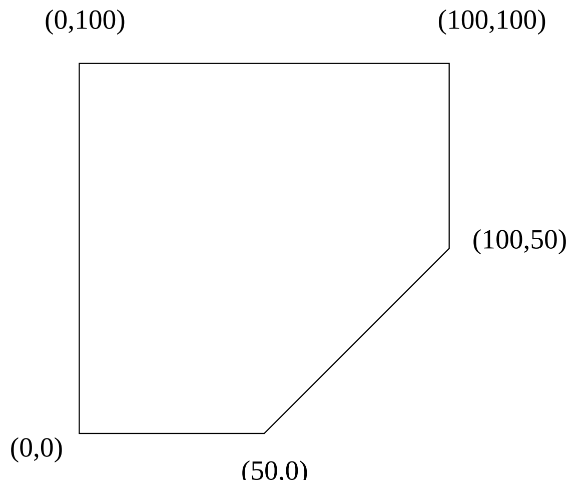

Figure 2: Square after clipping

Example 4 (polygon clipping continued)

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
```

Example 4 (polygon clipping continued)

The details of the polygon clipping algorithm are given in the paper. This example was included as an example of an array implementation of a recursive algorithm which examined (clipped) each line segment in the polygon on a line by line basis.

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.
- Array thinking discipline leads to more general solutions which may be used to solve other problems by changing the functions being applied.

Conclusions (continued)

- 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.

- Array languages, such as J, have great potential for implementation on parallel machines so that parallel algorithms may be written without much thought about parallelism beyond array thinking.

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.