F2SAD - prediction capabilities

van Halderen, A.W.; de Ronde, J.F.; Sloot, P.M.A.
Commission of the European Communities

****************

ESPRIT III

PROJECT NB 6756

****************

CAMAS

COMPUTER AIDED MIGRATION OF APPLICATIONS SYSTEM

****************

CAMAS-TR-2.2.4.6
F2SAD - prediction capabilities

****************

Date: March 1995 — Review 5.0

ACE - Univ. of Amsterdam - ESI SA - ESI GmbH - FEGS - PARSYTEC - Univ. of Southampton
Authors: Berry A.W. van Halderen  
Jan de Ronde  

March, 1995  
University of Amsterdam,  
Faculty of Mathematics and Computer Science  
Parallel Scientific Computing and Simulation group  
Netherlands
Chapter 1
Introduction

This report describes an evaluation of the F2SAD tool with several well known basic algorithms. In this report we consider some sorting algorithms. With this report we respond to the specific request of the review commission to make a more detailed validation of the prediction capabilities of the UvA workbench tools.

The intention of this document is to provide confidence in the ability of the tools (that implement the models developed in SAD and PARASOL) to estimate the execution time of some well known algorithms. The algorithms described here have a performance behaviour that is common knowledge. Despite this fact, we are still able to come up with some points that are interesting and not plain textbook knowledge.

In this report, various figures depicting measured and predicted execution times of Fortran programs will pass. The main part of the annotated algorithms has been included in appendix A.

At the time of the CAMAS review 5 we will present additional results on numerical relaxation algorithms.
Chapter 2
Sorting algorithms

Sorting algorithms are symbolic algorithms. Rather than performing heavy computations they compare and manipulate (swap or reorder) data. Sorting algorithms are quite well understood in their complexity behaviour. Despite this fact, few textbooks do actually compare the execution time of the algorithms. Two algorithms can be of the same order of complexity and still differ in their performance because a different number of instructions is executed.

Figure 2.1: This figure shows pure times of the sorting algorithms. The algorithms are applied to an array of uniform random numbers. It has been included here only to give an indication of the real execution time, since it is clearly not very informative when comparing the algorithms. All other figures therefore include either a logarithmic vertical axis and/or the execution time divided by the number of elements. This last type of figure works well to view the scalability of an algorithm and the crossover points for selecting between algorithms, but the overhead introduced by any algorithm is rather obscured. The logarithmic vertical axis plots still show the overhead, but the crossover points and behaviour is less visible. When viewing the other figures one has to keep in mind this figure, which tells you that the execution time of the sorting algorithms of the same complexity are actually very similar.

If we classify the sorting algorithms by their average-case time complexity, we can distinguish three classes. The exponential order $O(n^2)$, the logarithmic based sorting algorithms $n \log(n)$ and the linear time sorting algorithms $O(n)$. The exponential order algorithms are sometimes still (unjustifiably) used if a programmer is lazy and works on small arrays or under the disguise of a parallel computer. Shell-sort is such a variant used on parallel computers because of its parallel nature.

The most common sorting algorithms are the $O(n \log n)$ order sorting algorithms like quick-, heap- and mergesort. Although all three have the same average-case order of
2.1 RANDOMLY DISTRIBUTED INPUT DATA

<table>
<thead>
<tr>
<th>algorithm</th>
<th>average</th>
<th>worst-case</th>
<th>implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubblesort</td>
<td>$O(n^2)$</td>
<td>$\Theta(n^2)$</td>
<td>array (list also possible)</td>
</tr>
<tr>
<td>selectsort</td>
<td>$O(n^2)$</td>
<td>$\Theta(n^2)$</td>
<td>array (list also possible)</td>
</tr>
<tr>
<td>quicksort</td>
<td>$O(n \log n)$</td>
<td>$\Theta(n^2)$</td>
<td>array</td>
</tr>
<tr>
<td>heapsort</td>
<td>$O(n \log n)$</td>
<td>$\Theta(n \log n)$</td>
<td>array</td>
</tr>
<tr>
<td>mergesort</td>
<td>$O(n \log n)$</td>
<td>$\Theta(n \log n)$</td>
<td>list</td>
</tr>
<tr>
<td>bucketsort</td>
<td>$O(n)$</td>
<td>$\Theta(n^2)$</td>
<td>list</td>
</tr>
</tbody>
</table>

Table 2.1: Sorting algorithms; their time complexity as average case $\Theta(\ldots)$ and worst-case time complexity $\Theta(\ldots)$. The implementation can either be in linear array or a linked list form.

performance, they are inherently different. Quicksort has very bad worst-case performance, namely $\Theta(n^2)$. It is almost impossible to implement mergesort on arrays and the natural way to implement heapsort is on arrays. It is controversial whether quicksort is the fastest (on average), mergesort or heapsort. Comparing these algorithms is an interesting test case for our tool, F2SAD.

The theoretical complexity limit to sorting is $O(n \log n)$, but also in this case the practical knowledge of constraints to the input of the sorting algorithms can be exploited. To this albeit linear order sorting algorithms have been developed. These include radix, counting and bucketsort, in which the latter is sometimes used as a classifier or find-algorithm. These sortings algorithms do not base their algorithm on comparisons, like the sorting algorithms earlier mentioned, but rather on classification or decision trees.

The predicted times in these sections were produced using F2SAD which will now also incorporate the Parasol II tool in the same program. The tool produces a time-complexity formula in which the machine constants and the algorithmic parameters are still abstract. When not specified, we have used a Sun Classic LX model as test machine.

Memory : 32 MB
Model : SPARCstation LX
At frequency : 50 MHz
CPU : Texas Instruments TMS390S10 (MicroSparc)
Data cache : 2 Kb blocksize=16 1-way associative
Instruction cache : 4 Kb blocksize=32 1-way associative

2.1 Randomly distributed input data

Figure 2.2 shows the measured execution timings of the exponential and $O(n \log n)$ performance as well as the predicted execution timing. The input to the sorting algorithms is uniformly randomly distributed. Also the parameters in the time complexity formula have been set in such a way that they reflect this condition.

The reason for taking random input data is obviously that sorting algorithms will than expose an almost average-case behaviour. Some algorithms have a worst-case behaviour which is of a different complexity order or, less dramatic, have different minor terms in the complexity formula.

How the parameters are set we will come to later, but first we have a look at the measured and estimated execution time and compare them. Figure 2.3 shows the expected and estimated execution time for the bucketsort algorithm, compared to a mergesort implementation, while figure 2.2 gives the same data for the other sorting algorithms.

Figure 2.4 shows the error of the predicted versus the expected value.
CHAPTER 2. SORTING ALGORITHMS RANDOMLY DISTRIBUTED INPUT DATA

Figure 2.2: Measured and predicted execution times for the select-, bubble-, quick-, heap- and mergesort under the constraint that the input data to the sorting program is randomly distributed. The actual numbers are given in table ??.

The figures on the left show the measured execution time, on the right is the predicted execution time.
2.1. RANDOMLY DISTRIBUTED INPUT DATA

CHAPTER 2. SORTING ALGORITHMS

Figure 2.3: Measured and predicted execution times for the bucketsort algorithm with different numbers of buckets (16, 256, 1024 and 16384) compared against the underlying mergesort algorithm. The input data to the sorting programs is assumed to be uniform randomly distributed. The actual numbers are in table ???. The figures on the left show the measured execution time, on the right is the predicted execution time. The upper two graphs plot the time against a logarithmic axis, the lower graphs show the execution time spend (on average) per element in the array.

Figure 2.4: The error as a percentage of the expected value for the data shown in figures 2.2 and 2.3.
2.2 Setting the parameters

For the algorithmic parameters—the number of times loops and conditions are taken—the F2SAD/Parasol II toolset has basically two ways actualizing. One is by using a profile files to determine the parameters and the other is by defining them by hand. The programs which have been analyzed in this report have been deliberately analyzed by hand. Below we give an example of theoretical complexity parameters for the selection sorting algorithm.

2.2.1 Selection Sort

In appendix A, the algorithm considered here, can be found. The number of times the outer most loop at line 9 is executed is clearly \( n - 1 \), the size of the array to be sorted. The inner loop at line 12 has different properties. In the first iteration of the outermost loop it is executed \( n - 2 \) times, the second time \( n - 3 \) times continuing until it is executed only once. This leads to the summation:

\[
\sum_{i=1}^{n-2} i = \frac{1}{2}(n - 2) \left( (n - 2) + 1 \right) = \frac{1}{2}(n - 1)(n - 2)
\]

Since the outer loop iterates \( n - 1 \) times, the inner loop will iterate \( \frac{1}{2}(n - 2) \) times on average.

The conditional on line 13 is true whenever an element \( a_i \) in the list \( a_0, a_1 \ldots a_n \) is smaller than all its predecessors \( (a_{i-1}, a_{i-2} \ldots a_0) \). For \( i = 0 \) the conditional is always true, for \( i = 1 \) with a uniform random list this will be \( \frac{1}{2} \), for \( i = 2 \) it will be \( \frac{1}{4} \). We will not go into any detail, but the the idea between this logic is that each predecessor has a probability of \( \frac{1}{2} \) to be smaller and in this way each predecessor will half the chance that the element \( a_i \) is smaller than all its predecessors.

Each element \( a_i \) will be subject to the conditional \( i \) times, which leads to the following formula for the chance that the condition evaluates to true:

\[
\frac{\frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{n-i}}{n-i} = \frac{1}{2} \ln(n) + C
\]

The nominator approaches 2 thus we get:

\[
\sum_{i=0}^{n-1} \frac{2}{n-i} = 2 \ln(n) + C
\]

In which we can ignore \( C \), and since there are \( n \) elements we have to divide this by \( n \)

To recapulate we are left with the following parameters:

- \( N.1 \ n-1 \) (the outer loop)
- \( N.2 \ 0.5*(n-2) \) (the inner loop)
- \( P.1 \ 2*\ln(n)/n \) (the constant)
2.3. A NOTE ON LINEAR SORTING  

As was mentioned above the linear order sorting algorithms use some other sorting algorithm to sort the classes they have built. As we have seen, the overhead and the usage of an other algorithm do not make it attractive for sorting purposes, since it is only very slightly better than the underlying sort. But, the linear order sorting algorithms have also a very different purpose. If it is necessary to classify the input in ranges, resulting in a list of only roughly sorted lists, there is no need for the underlying comparison sort mechanism. And therfore these algorithms have their separate usefulness, especially in parallel computers in which data has be redistributed. The bucket method can be used to classify the data, and to distribute each class to a processor.

We have used here the notation $N.x$ and $P.x$, for the control flow parameters, which is also used by F2SAD. The numbers after the $N.$ and $P.$ have no real meaning, but are distributed according to the flow of the program. They are the same each time the program is run through F2SAD.

For sorted data the conditional $P.2$ will be nearly 0 (actually it will be $\frac{n}{n-1}(n-2)$, all other parameters remain the same.

Figure 2.5 gives results in the hypothetical case that all input data is sorted. In that case obviously for example the bubblesort algorithm displays a very friendly execution time behaviour.

2.3 A note on linear sorting

Figure 2.5: For already sorted input data.
Appendix A

Source code

This appendix includes all the source code of the algorithms studied in this report. The main program is not included since it is generated in order to provide multiple input data sets.

A.1 Bubblesort

```fortran
SUBROUTINE bubblesort(asize, a)
IMPLICIT NONE
INTEGER asize
DOUBLE PRECISION a
DIMENSION a(*)
DOUBLE PRECISION swap
INTEGER i, size
LOGICAL flag

size = asize
flag = .FALSE.
DO 20, i=1, size-1
   IF(a(i) .GT. a(i+1)) THEN
      PRINT *, i, a(i), a(i+1)
      swap = a(i)
      a(i) = a(i+1)
      a(i+1) = swap
      flag = .TRUE.
   END IF
20 CONTINUE
size = size - 1
IF(flag) GOTO 10
END
```
A.2. Selectsort

SUBROUTINE selectsort(size, a)
IMPLICIT NONE
INTEGER size
DOUBLE PRECISION a
DIMENSION a(*)
DOUBLE PRECISION smallest
INTEGER i, j, index

DO 20, i=1, size-1
   index = i
   smallest = a(index)
   DO 10, j=i+1, size
      IF(smallest .GT. a(j)) THEN
         index = j
         smallest = a(index)
      END IF
   10 CONTINUE
   a(index) = a(i)
   a(i) = smallest
20 CONTINUE
END
A.3 Heapsort

The heapify routine is the key to the heapsort algorithm. The parameters to the heapify routine are an array A and an index i into that array. The precondition for the heapify routine is that the left binary subtree and the right binary subtree are both heaps. A(i) however may be larger than the elements in both subtrees, thus violating the heap property. The heapify routine will “sift down” this element A(i) and by this way both subtrees and A(i) will become one larger heap.

\[
\begin{align*}
&i \leftarrow \text{Left}(i) \\
r \leftarrow \text{Right}(i) \\
&\text{if } l \leq \text{HeapSize}[A] \text{ and } A[l] > A[i] \\
&\quad \text{then } \text{largest} \leftarrow l \\
&\quad \text{else } \text{largest} \leftarrow i \\
&\text{if } r \leq \text{HeapSize}[A] \text{ and } A[r] > A[\text{largest}] \\
&\quad \text{then } \text{largest} \leftarrow r \\
&\text{if } \text{largest} \neq i \\
&\quad \text{then } \text{exchange } A[i] \leftrightarrow A[\text{largest}] \\
&\quad \text{Heapify}(A[\text{largest}])
\end{align*}
\]

Most parameters of the heapsort are

\[
\begin{align*}
&\text{a}(i) = a(\text{largest}) \\
&\text{a}(\text{largest}) = \text{swap} \\
&i = \text{largest} \\
&\text{GOTO } 10 \\
&\text{END IF}
\end{align*}
\]
for $i \leftarrow \frac{\text{Size}[A]}{2}$ downto 1  
    do Heapify($A[i]$)  

SUBROUTINE buildheap(size, a)  
IMPLICIT NONE  
DOUBLE PRECISION a  
INTEGER size  
DIMENSION a(*)  
INTEGER i  

DO 10, i=size/2, 1, -1  
    CALL heapify(size, a, i)  
10 CONTINUE  

END  

BuildHeap(A)  
for $i \leftarrow \text{length}[A]$ downto 2  
    do exchange $A[1] \leftrightarrow A[i]$  
        decrease HeapSize by 1  
        Heapify(A,1)  

SUBROUTINE heapsort(asize, a)  
IMPLICIT NONE  
INTEGER asize  
DOUBLE PRECISION a  
DIMENSION a(*)  
DOUBLE PRECISION swap  
INTEGER i, size  

size = asize  
CALL buildheap(size, a)  
DO 10, i=size, 2, -1  
    swap = a(1)  
    a(1) = a(i)  
    a(i) = swap  
    size = size - 1  
    CALL heapify(size, a, 1)  
10 CONTINUE  

END
A.4 Mergesort

SUBROUTINE mergelsort(a, lstptr, head, tail)
IMPLICIT NONE
INTEGER stacksize
PARAMETER (stacksize = 256)
INTEGER lstptr, head(*), tail(*)
DOUBLE PRECISION a(*)
INTEGER stackindex, stack(stacksize)
INTEGER size, list1, list2, run, hsize, x, y, z

x = 0
y = 0
z = 0

size = 0
run = lstptr
10 IF(run .GT. 0) THEN
    size = size + 1
    run = tail(run)
    GOTO 10
END IF
sv = size
23 stack(1) = 0
24 stackindex = 2
1 IF(size .LE. 1) GO TO 4
list1 = lstptr
list2 = lstptr
hsize = size/2
20 IF(hsize .GT. 0) THEN
    hsize = hsize - 1
    list2 = tail(list2)
    GOTO 20
END IF
2 stack(index) = size
2 stack(index+1) = list2
2 stack(index+2) = 1
stackindex = stackindex + 3
size = size/2
lstptr = list1
GO TO 1
2 list1 = lstptr
stackindex = stackindex - 3
size = stack(stackindex)
list2 = stack(stackindex+1)
49 stack(stackindex) = size
stack(stackindex+1) = list1
stack(stackindex+2) = 2
stackindex = stackindex + 3
size = size - size/2
lstptr = list2
GO TO 1
3 list2 = lstptr
stackindex = stackindex - 3
size = stack(stackindex)
list1 = stack(stackindex+1)
IF(a(head(list1)) .LT. a(head(list2))) THEN
    lstptr = list1
ELSE
    lstptr = list2
END IF

run = 0
30 IF(list1 .GT. 0 .AND. list2 .GT. 0) THEN
    z = z + 1
    IF(a(head(list1)) .LT. a(head(list2))) THEN
        IF(run .GT. 0) THEN
            tail(run) = list1
        ELSE
            lstptr = list1
        END IF
        run = list1
        list1 = tail(list1)
    ELSE
        IF(run .GT. 0) THEN
            tail(run) = list2
        ELSE
            lstptr = list2
        END IF
        run = list2
        list2 = tail(list2)
    END IF
GOTO 30
END IF
IF(list1 .GT. 0) THEN
    IF(run .GT. 0) THEN
        tail(run) = list1
    ELSE
        lstptr = list1
    ENDIF
ELSE IF(list2 .GT. 0) THEN
    IF(run .GT. 0) THEN
        tail(run) = list2
    ELSE
        lstptr = list2
    END IF
END IF
END IF
IF(list1 .GT. 0) THEN
    IF(run .GT. 0) THEN
        tail(run) = list1
    ELSE
        lstptr = list1
    ENDIF
ELSE IF(list2 .GT. 0) THEN
    IF(run .GT. 0) THEN
        tail(run) = list2
    ELSE
        lstptr = list2
    END IF
END IF

F2C isn’t able to process vector-if statements, that is why the following IF-statement is commented out and two replacement IF’s are dropped in.

4    IF(size .EQ. 1) tail(lstptr) = 0
y = y + 1
IF(stack(stackindex-1).EQ.1) GO TO 2
x = x + 1
IF(stack(stackindex-1).EQ.2) GO TO 3
END
A.5 Quicksort

SUBROUTINE quicksort(asize, a)
INTEGER stacksize
PARAMETER (stacksize = 256)
INTEGER asize
DOUBLE PRECISION a
DIMENSION a(*)
DOUBLE PRECISION aux
INTEGER start, size, front, back, stack, idx
DIMENSION stack(stacksize)

size = asize
idx = 0
start = 1
20 IF(size .GT. 1) THEN
    front = start+1
    back = start+size-1
30 IF(front .LE. back) THEN
    IF (a(start) .GT. a(front)) THEN
        front = front + 1
    ELSE IF(a(start) .LE. a(back)) THEN
        back = back - 1
    ELSE
        aux = a(front)
        a(front) = a(back)
        a(back) = aux
    END IF
    GOTO 30
ENDIF
31 IF(front .NE. start+1) THEN
    aux = a(start)
    a(start) = a(front-1)
    a(front-1) = aux
    stack(idx+1) = front - start - 1
    stack(idx+2) = start
    idx = idx + 2
ENDIF
32 size = size - back + start - 1
33 start = front
GOTO 20
ENDIF
34 IF(idx .GT. 0) THEN
    idx = idx - 2
    size = stack(idx+1)
    start = stack(idx+2)
GOTO 20
ENDIF
END
A.6. **BUCKETSORT**

APPENDIX A. SOURCE CODE

A.6  **BUCKETSORT**

1
2    SUBROUTINE bucketlsort(a, lstptr, head, tail)
3    IMPLICIT NONE
4    DOUBLE PRECISION a(*)
5    INTEGER lstptr, head(*), tail(*)
6    INTEGER numbuckets
7    PARAMETER (numbuckets = 16384)
8    INTEGER buckets(numbuckets), bucketnum, i, aux
9
10   DO 10, i=1, numbuckets
11      buckets(i) = 0
12   10 CONTINUE
13   20 IF(lstptr .GT. 0) THEN
14      bucketnum = INT(a(head(lstptr)) * numbuckets) + 1
15      aux = tail(lstptr)
16      tail(lstptr) = buckets(bucketnum)
17      buckets(bucketnum) = lstptr
18      lstptr = aux
19      GO TO 20
20   END IF
21
22   DO 30, i=1, numbuckets
23      CALL mergelsort(a, buckets(i), head, tail)
24   30 CONTINUE
25
26   lstptr = 0
27   DO 40, i=1, numbuckets
28      IF(buckets(i) .GT. 0) THEN
29         IF(lstptr .EQ. 0) THEN
30            lstptr = buckets(i)
31         ELSE
32            tail(aux) = buckets(i)
33         END IF
34         aux = buckets(i)
35      50 IF(tail(aux)) THEN
36         aux = tail(aux)
37         GO TO 50
38      END IF
39   40 CONTINUE
40
41   END