

# **SANDIA REPORT**

SAND93-0933

Unlimited Release

Printed December 1998

## **Optimizing the Point-in-Box Search Algorithm for the Cray Y-MP™ Supercomputer**

M. E. Davis, M. W. Heinstein, S. W. Attaway, and J. W. Swegle

Prepared by  
Sandia National Laboratories  
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,  
a Lockheed Martin Company, for the United States Department of  
Energy under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.



**Sandia National Laboratories**

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from  
National Technical Information Service  
U.S. Department of Commerce  
5285 Port Royal Rd  
Springfield, VA 22161

NTIS price codes  
Printed copy: A03  
Microfiche copy: A01



SAND93-0933  
Unlimited Release  
Printed December 1998

## **Optimizing the Point-in-Box Search Algorithm for the Cray Y-MP<sup>TM</sup> Supercomputer\***

M. E. Davis  
Cray Research Inc.  
6565 America's Pkwy. NE #830  
Albuquerque, NM 87111

M.W. Heinstein, S.W. Attaway, J.W. Swegle  
Engineering and Manufacturing Mechanics Department  
Sandia National Laboratories  
P.O. Box 5800  
Albuquerque, NM 87185-0443

### **Abstract**

Determining the subset of points (particles) in a problem domain that are contained within certain spatial regions of interest can be one of the most time-consuming parts of some computer simulations. Examples where this "point-in-box" search can dominate the computation time include (1) finite element contact problems; (2) molecular dynamics simulations; and (3) interactions between particles in numerical methods, such as discrete particle methods or smooth particle hydrodynamics. This paper describes methods to optimize a point-in-box search algorithm developed by Swegle [1] that make optimal use of architectural features of the Cray Y-MP<sup>TM</sup> Supercomputer.

\*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000

# 1 Introduction

The current generation of numerical analysis codes, such as the Finite Element Method (FEM), Discrete Motion Codes (DMC), Smooth Particle Hydrodynamics (SPH), and Particle In Cell Methods (PIC), are increasingly solving larger problems. Often, one of the most time-consuming parts of the analysis involves the determination of which points in the problem domain are contained within certain spatial regions of interest. Since a region of interest can be expressed most efficiently in the form of a box or cube, the problem becomes one of finding which of a given set of points lies inside the box.

Swegle [1] describes a search algorithm for solving the problem of determining which of a given set of points lies inside a box. It involves constructing an ordered list of points by sorting the points on each rectangular coordinate value and then searching for the points that lie within the box. The search algorithm is economical in its use of memory. In 3D, for example, the algorithm requires only  $7N$  memory locations, where  $N$  is the number of points. The binary search technique used in [1] for finding the subset of points in the box is a classical method for solving a search problem of this kind on a conventional sequential computer.

This paper describes several ways in which optimizations can be done on a supercomputer, such as the Cray Y-MP, to achieve greater efficiency and lower cost for this kind of computation. Section 2 reviews the search algorithm developed by Swegle. Section 3 describes a general-purpose proprietary software library routine on the Y-MP for optimizing the binary search. Section 4 describes a technique used to optimize a set of searches for the particular problem at hand. Section 5 describes a way to minimize the number of indirect memory references. Section 6 covers test results for two different point sets.

## 2 Swegle Search Algorithm

This section reviews the point-in-box search algorithm developed by Swegle [1]. Briefly the algorithm consists of individual one-dimensional sorts of the points using each coordinate value as the search key, followed by binary searches of each sorted list to find the points at the edge of the search region (box). This produces separate sets of points whose positions fall within the bounds of the box for each coordinate direction. Finally the three lists (two lists for a 2D problem) are intersected to obtain the points inside the box. Figure 1 shows a schematic of the three steps in the algorithm. A description of each step is given below, followed by a detailed example.

### 2.1 Sort

The sorting step constructs an ordered list of the points for each coordinate direction. The result of this sort is an index vector for each coordinate,  $\{I_x, I_y, I_z\}$ , that contains the point IDs in order of increasing coordinate value. One additional set of vectors,  $\{R_x, R_y, R_z\}$ , called the rank vectors is also constructed. It gives the location of each point in the index vector. It can be easily constructed by looping through the index vector. For example, suppose point  $j$  is stored at position  $i$  in the index vector; then the rank vector would store the pointer  $i$  at its position  $j$ . The rank vector is required to avoid searching the index vector for a given point. The memory requirements for this step is  $2*N*ND$ , where  $N$  is the number of points and  $ND$  is the spatial dimension of the point set.

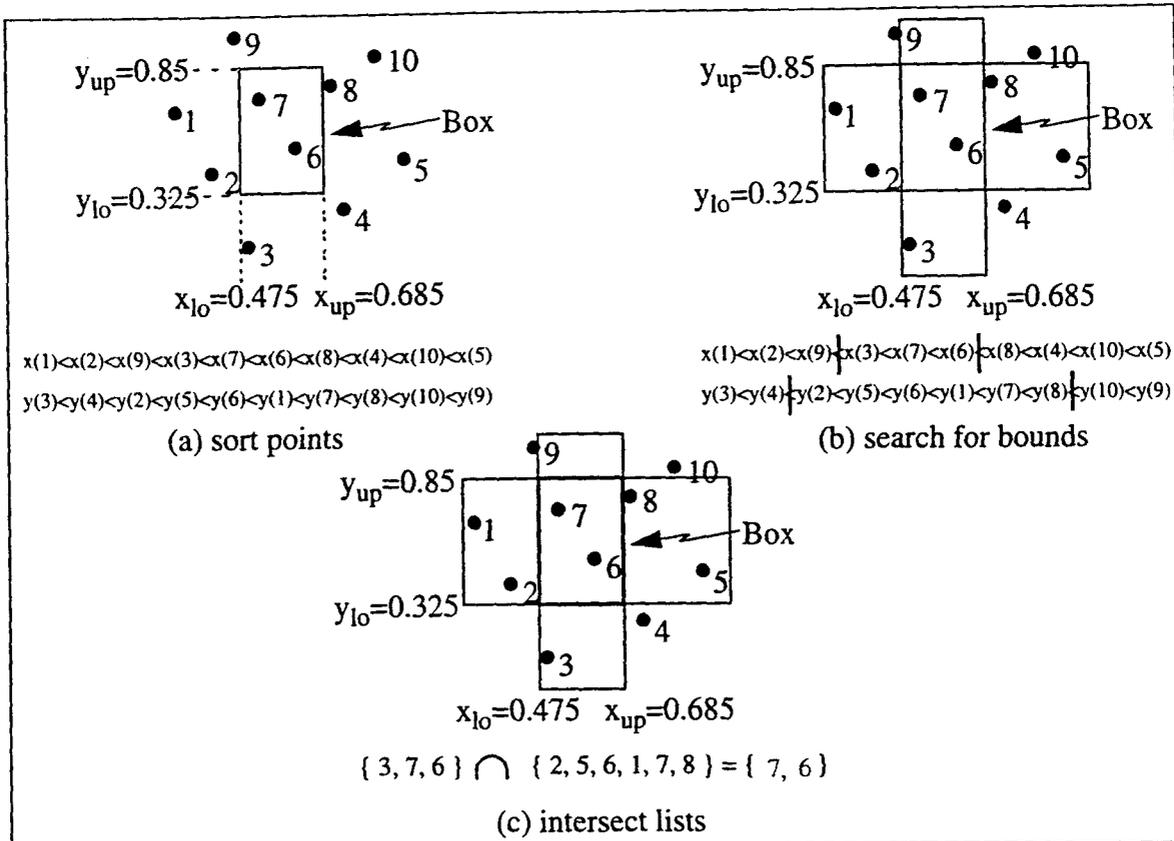


Figure 1. Schematic of Swegle Search Algorithm

## 2.2 Search

The second step is to form three lists (one for each coordinate) which contain those points that are within the minimum and maximum bounds of the box. Each list is formed using two binary searches on the index vector: one to find the pointer corresponding to the first point inside the box, and the other to find the pointer corresponding to the last point within the box (the lists are therefore never formed - instead it is the 'lo' pointers and the 'up' pointers that are stored). Figure 2 shows one step in the  $O(\log_2 N)$  binary search, where the target  $x_{\text{target}}$  corresponds to one of the bounds  $x_{\text{min}}$  or  $x_{\text{max}}$  of the box.

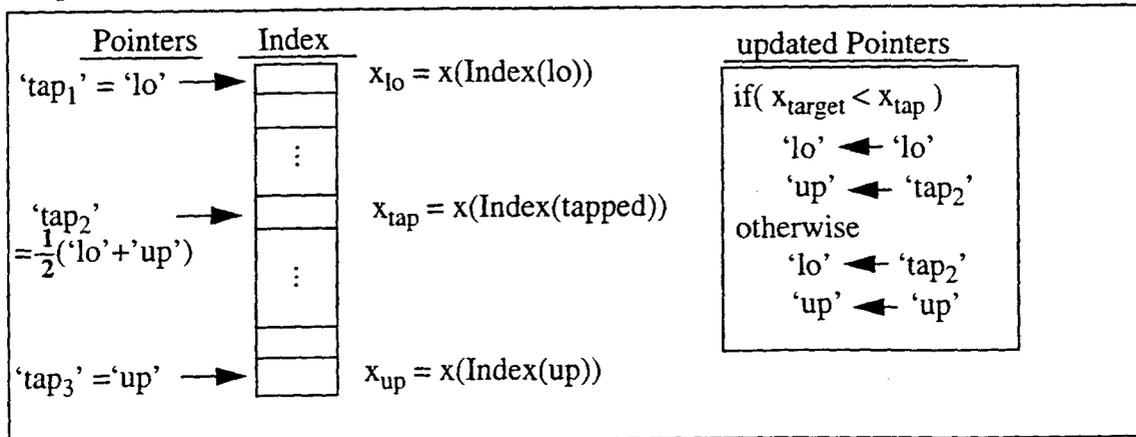


Figure 2. Classical binary search step to find a pointer into index array

The binary search continues until 'lo' + 1 = 'up' and  $x_{lo} < x_{target} < x_{up}$ . Upon completing the search, the pointer into the index vector is then  $(i_x)_{min} = 'up'$  for the target  $x_{target} = x_{min}$ , and  $(i_x)_{max} = 'lo'$  for the target  $x_{target} = x_{max}$ .

### 2.3 Intersection

Finally, the lists are intersected to find the points in the box for each coordinate simultaneously. To accomplish this, each of the points in one list is selected and then checked if its rank is between the lower and upper pointer in the other two coordinates. For computational efficiency the shortest list of points is selected, which can be determined by selecting the smallest of  $[(i_w)_{max} - (i_w)_{min} + 1]$ ,  $w = x, y, \text{ or } z$ . Suppose, for example, that the list for the y-coordinate contains the smallest number of points. Then the points in this list

$$i = I_y((i_y)_{min}), I_y((i_y)_{min} + 1), \dots, I_y((i_y)_{max}) \text{ are in the box if}$$

$$(i_x)_{min} \leq R_x(i) \leq (i_x)_{max} \text{ and } (i_z)_{min} \leq R_z(i) \leq (i_z)_{max} .$$

### 2.4 Example Problem

The 2D example problem shown in Figure 1 is used to illustrate the algorithm. The problem consists of finding which of the  $N=10$  points lie inside of the box defined by:  $x_{min}=0.475$ ,  $x_{max}=0.685$  and  $y_{min}=0.325$ ,  $y_{max}=0.85$ . The index vectors are:

(Index) <sub>x</sub>	1	2	9	3	7	6	8	4	10	5
(Index) <sub>y</sub>	3	4	2	5	6	1	7	8	10	9

This indicates, for example, that along the x-coordinate direction, point 1 has the smallest x-coordinate, point 2 the next smallest, point 9 the next, and so on. The rank vectors give the location of each point in the index vectors and are:

(Rank) <sub>x</sub>	1	2	4	8	10	6	5	7	3	9
(Rank) <sub>y</sub>	6	3	1	2	4	5	7	8	10	9

The search step finds two lists of points. Each list is formed using a binary search on its respective index vector to find the pointers that correspond to the points that are just on or inside of the bounds ( $x_{min}, x_{max}$  and  $y_{min}, y_{max}$ ). Figure 3, for example, shows the binary search to find the pointer corresponding to the point just greater than or equal to  $x_{min}=0.475$ .

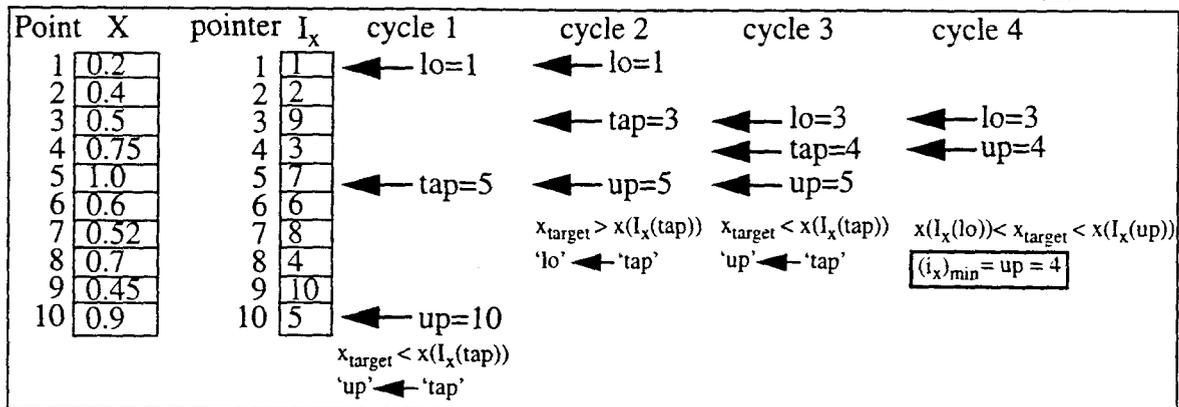


Figure 3. Binary search to find point just greater than or equal to  $x_{min}=0.475$

For the example problem, the binary search would result in the following pointers:  $(i_x)_{\min} = 4$ ,  $(i_x)_{\max} = 6$ ,  $(i_y)_{\min} = 3$ ,  $(i_y)_{\max} = 8$ , so that point 3 is the first point greater than or equal to  $x_{\min}$  and point 6 is the last point less than or equal to  $x_{\max}$ , etc.

Finally, the two lists are intersected to find the points in the box. The list for the x-coordinate is the one with the smallest number of points, so that the points

$$i = I_x((i_x)_{\min}), I_x((i_x)_{\min} + 1), \dots, I_x((i_x)_{\max}) = 3, 6, 7 \quad \text{are in the box if} \\ (i_y)_{\min} \leq R_y(i) \leq (i_y)_{\max} .$$

To help illustrate this procedure, the three points in the smallest list are processed as follows:

$$1) \quad i = I_x((i_x)_{\min}) = 3 \quad \text{and} \quad R_y(3) = 1.$$

Since  $1 < (i_y)_{\min}$ , point  $i=3$  is **not** in the box.

$$2) \quad i = I_x((i_x)_{\min} + 1) = 7 \quad \text{and} \quad R_y(7) = 7.$$

Since  $(i_y)_{\min} \leq 7 \leq (i_y)_{\max}$  point 7 is in the box.

$$3) \quad i = I_x((i_x)_{\min} + 2) = 6 \quad \text{and} \quad R_y(6) = 5.$$

Since  $(i_y)_{\min} \leq 5 \leq (i_y)_{\max}$  point 6 is also in the box.

After intersecting the lists, it is found that only the points  $\{6, 7\}$  are in the box.

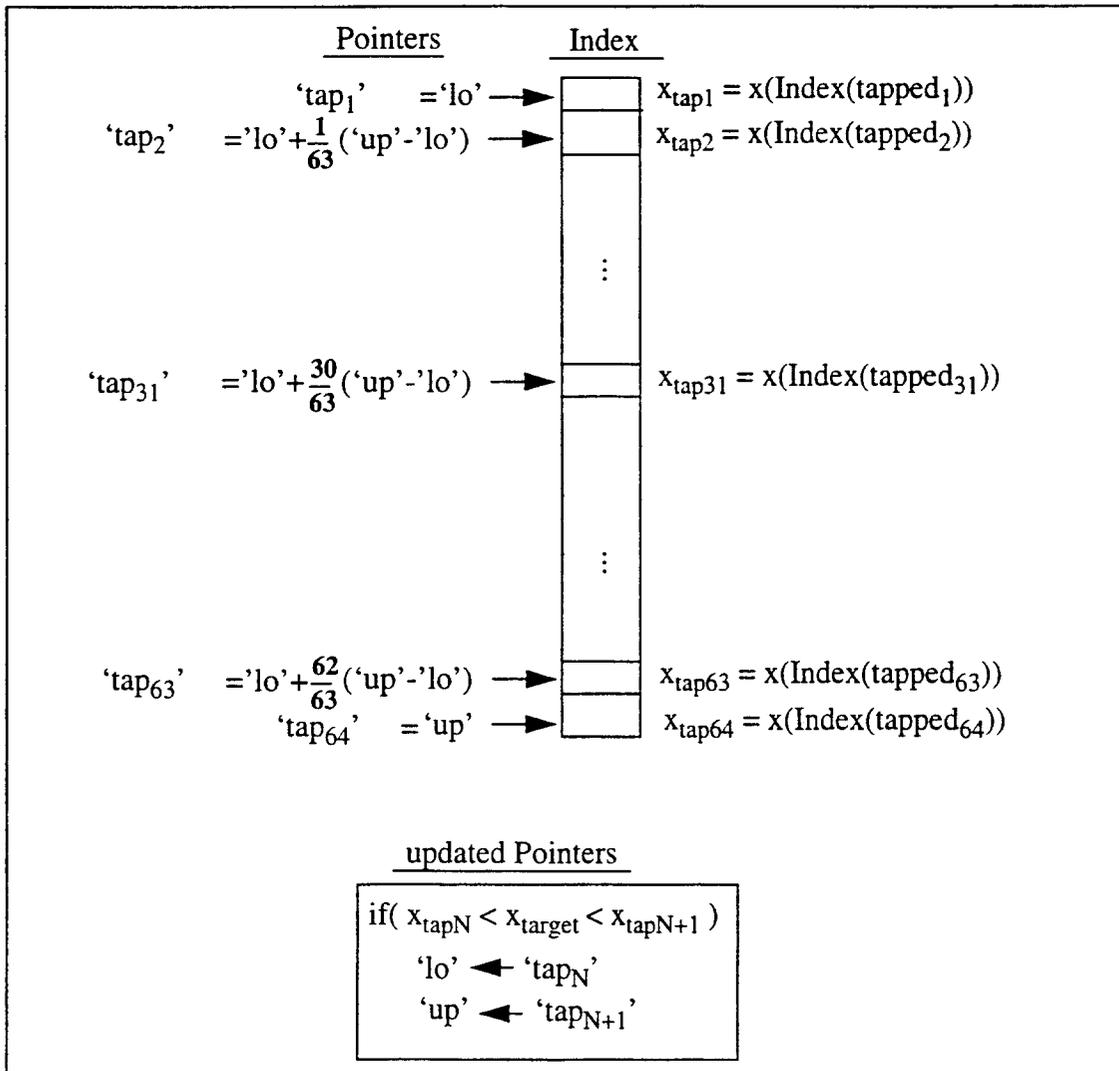
## 2.5 Summary and Motivation for Current Work

The great advantage of this algorithm is that it is nearly independent of the geometry of the point set and is very economical in its use of memory. In 3D, for example, the search algorithm requires only  $(7N)$  memory locations, where  $N$  is the number of points. This work is aimed at improving the speed of the algorithm.

In the following sections, several methods to speed-up the Swegle search algorithm are described. These methods make optimal use of architectural features of the CRAY YMP supercomputer. Sections 3 and 4 describe two methods to optimize the binary search. Section 5 describes a method for optimally intersecting the lists.

### 3 Cray Research $\log_{64}N$ Binary Search

Proprietary general-purpose libraries on supercomputer systems such as the Cray Y-MP provide an optimized subroutine `osrchf` to solve the binary search problem [2]. In these routines, the architecture of the computer is exploited to the fullest extent possible so that the problem can be solved efficiently. On the Cray for example, a 64-element list of data can be processed as an atomic unit, so that a “binary” search on the Cray could be done in  $\log_{64}N$  time rather than in the  $\log_2N$  time that is required for scalar architectures -- a factor of 6 performance improvement in the search. This architecture-dependent binary search is accomplished by picking not a single element (“tap”) from the middle of the set of data between ‘lo’ and ‘up’, but by picking 64 elements at equally-spaced intervals. It is then possible to do the comparison of the target with the vector of “tapped” elements, as shown in Figure 2.



**Figure 4.**  $\log_{64}N$  binary search on the CRAY Y-MP

## 4 Vectorized Binary Search

The popular estimate of the maximum performance improvement to be gained by processing data in vector mode on the Cray Y-MP is 10 times scalar performance [3]. Given this, even the optimized proprietary library routine `osrchf`, running at 6 times the scalar rate, is not providing peak performance for a binary search of a single element. Realizing that the problem consists of finding the set of points that lie within each of many boxes, there is a way to achieve closer to peak performance by processing several of the boxes at once. This method would take as input the entire set of points to be searched, and a set of box endpoints (targets) to search for. It would then return a list of indexes, equal in size to the list of targets, indicating where in the point set the targets (would) reside. Clearly, this method requires more memory than the classical search for a single target.

Simply passing more data to the routine is not sufficient to get the routine to run faster. The basic problem with the classical binary search algorithm is that the binary search loop does not vectorize, as shown in Figure 5.

```
1SRCH PAGE 1 CRAY FORTRAN CFT77 5.0.0.0 06/12/91 14:30:52
  1 1. subroutine srch
  2 2. parameter (n=10000)
  3 3. common /ccml/ list,tgt,ind
  4 4. real list(n)
  5 5. ilo=1
  6 6. ihi=n
  7 7. log2n=64-leadz(n)+1
  8 8. do i=1,log2n
  9 9. itap=ilo+ihi/2
 10 10. if (tgt.gt.list(itap)) then
 11 11. ilo=itap
 12 12. else
 13 13. ihi=itap
 14 14. endif
 15 15. enddo
 16 16. ind=itap
 17 17. end

VECTORIZATION INFORMATION
-----
cft77-8009 cft77: VECTOR SRCH, Line = 8, File = srch.f, Line = 8

Loop starting at line 8 was not vectorized. A value defined conditionally is used in
another block.
```

Figure 5. Classical binary search loop does not vectorize

There are two key ideas that must be used in changing the binary search algorithm so that it will run in vector mode. The first is that only inner loops vectorize. This means that the classical binary search loop which iterates over the space of the point set must be interchanged with the one that iterates over the space of the target list. The other key idea is that the binary search loop always iterates the same number of times, namely  $1+\log_2 N$ , where  $N$  is the number of elements in the point set. This is true no matter what the value of the target is. Most implementations of a binary search loop use a test of the target and the "tap" as the loop control condition. This kind of test is too irregular to work easily as an outer loop, but a loop

that iterates from 1 to  $1+\log_2 N$  is quite workable. Armed with these ideas, a vectorized  $\log_2 N$  binary search algorithm can be implemented, as shown in Figure 6.

```

IOSRCHFRM                                CRAY FORTRAN CFT77 5.0.0.0 06/12/91 14:30:52
 1  1. c
 2  2. c-----
 3  3. c
 4  4.      subroutine osrchfrm (nx,x,indord,nt,tgtmin,tgtmax,imin,imax)
 5  5. c
 6  6. c
 7  7. c
 8  8. c
 9  9. c produce in each imin(i) [imax(i)] the index into the integer
10 10. c array indord whose referenced element in the real array x
11 11. c has the smallest [largest] value greater than [less than] that
12 12. c of tgtmin(i) [tgtmax(i)].
13 13. c
14 14.      implicit none
15 15. c
16 16. c inputs:
17 17. c
18 18.      integer nx ! number of elements in array x
19 19.      real x(nx) ! array of real numbers
20 20.      integer indord(nx) ! array of indexes into x, from ORDERS
21 21.      integer nt ! number of elements in the target arrays
22 22.      real tgtmin(nt) ! array of minimum targets to be found
23 23.      real tgtmax(nt) ! array of maximum targets to be found
24 24. c
25 25. c outputs:
26 26. c
27 27.      integer imin(nt) ! indexes into x of min targets
28 28.      integer imax(nt) ! indexes into x of max targets
29 29. c
30 30. c scratch:
31 31. c
32 32.      integer ntaps ! number of taps into x required
33 33.      integer i ! loop index over nt
34 34.      integer lo(nt) ! current low tap
35 35.      integer hi(nt) ! current high tap
36 36.      integer j ! loop index over ntaps
37 37.      integer tap(nt) ! current middle tap
38 38. c
39 39.      ntaps = 64-leadz(nx) ! log base 2 of nx + 1
40 40. c
41 41. c first the min
42 42. c
43 43.      lo = 1
44 44.      hi = nx
45 45.      do j = 1,ntaps
46 46.          tap = (lo+hi)/2
47 47.          do i = 1,nt
48 48.              if (x(indord(tap(i))).ge.tgtmin(i)) then
49 49.                  hi(i) = tap(i)
50 50.              else
51 51.                  lo(i) = tap(i)
52 52.              end if
53 53.          end do
54 54.      end do

```

Figure 6. Vectorized  $\log_2 N$  binary search on the CRAY Y-MP

```

55 55.      do i = 1,nt
56 56.          if (x(indord(lo(i))).ge.tgtmin(i)) then
57 57.              imin(i) = lo(i)
58 58.          else
59 59.              imin(i) = hi(i)
60 60.          end if
61 61.      end do
62 62.  c
63 63.  c now the max
64 64.  c
65 65.      lo = 1
66 66.      hi = nx
67 67.      do j = 1,ntaps
68 68.          tap = (lo+hi)/2
69 69.          do i = 1,nt
70 70.              if (x(indord(tap(i))).le.tgtmax(i)) then
71 71.                  lo(i) = tap(i)
72 72.              else
73 73.                  hi(i) = tap(i)
74 74.              end if
75 75.          end do
76 76.      end do
77 77.      do i = 1,nt
78 78.          if (x(indord(hi(i))).le.tgtmax(i)) then
79 79.              imax(i) = hi(i)
80 80.          else
81 81.              imax(i) = lo(i)
82 82.          end if
83 83.      end do
84 84.      end

```

VECTORIZATION INFORMATION

```

-----
cft77-8004 cft77: VECTOR OSRCHFRM, Line = 43, File = osrchfrm.f, Line = 43
Loop starting at line 43 was vectorized.
cft77-8004 cft77: VECTOR OSRCHFRM, Line = 44, File = osrchfrm.f, Line = 44
Loop starting at line 44 was vectorized.
cft77-8035 cft77: VECTOR OSRCHFRM, Line = 45, File = osrchfrm.f, Line = 45
Loop starting at line 45 was not vectorized. It contains an inner loop.
cft77-8004 cft77: VECTOR OSRCHFRM, Line = 46, File = osrchfrm.f, Line = 46
Loop starting at line 46 was vectorized.
cft77-8004 cft77: VECTOR OSRCHFRM, Line = 47, File = osrchfrm.f, Line = 47
Loop starting at line 47 was vectorized.
cft77-8004 cft77: VECTOR OSRCHFRM, Line = 55, File = osrchfrm.f, Line = 55
Loop starting at line 55 was vectorized.
cft77-8004 cft77: VECTOR OSRCHFRM, Line = 65, File = osrchfrm.f, Line = 65
Loop starting at line 65 was vectorized.
cft77-8004 cft77: VECTOR OSRCHFRM, Line = 66, File = osrchfrm.f, Line = 66
Loop starting at line 66 was vectorized.
cft77-8035 cft77: VECTOR OSRCHFRM, Line = 67, File = osrchfrm.f, Line = 67
Loop starting at line 67 was not vectorized. It contains an inner loop.
cft77-8004 cft77: VECTOR OSRCHFRM, Line = 68, File = osrchfrm.f, Line = 68
Loop starting at line 68 was vectorized.
cft77-8004 cft77: VECTOR OSRCHFRM, Line = 69, File = osrchfrm.f, Line = 69
Loop starting at line 69 was vectorized.
cft77-8004 cft77: VECTOR OSRCHFRM, Line = 77, File = osrchfrm.f, Line = 77
Loop starting at line 77 was vectorized.

```

Figure 6. (cont'd)

## 5 Direct List Intersection

The final step in the search algorithm is to intersect the three lists to find the points in the box for each coordinate simultaneously. To accomplish this, each of the points in one list is selected and then checked if its rank is between the lower and upper pointer in the other two coordinates. For computational efficiency the shortest list of points is selected, which can be determined by selecting the smallest of  $[(i_w)_{\max} - (i_w)_{\min} + 1]$ ,  $w = x, y, \text{ or } z$ . A straightforward implementation of the list intersection is shown in Figure 7.

```
c select the smallest list
  num1 = iup(1) - ilo(1) + 1
  num2 = iup(2) - ilo(2) + 1
  num3 = iup(3) - ilo(3) + 1

  if( num1 .le. num2 .and. num1 .le. num3 )then
    ixyz = 1
    iy = 2
    iz = 3
    num = num1
  else if( num2 .le. num1 .and. num2 .le. num3 )then
    ixyz = 2
    iy = 1
    iz = 3
    num = num2
  else
    ixyz = 3
    iy = 1
    iz = 2
    num = num3
  endif

c first intersection
  do 101 i1 = ilo(ixyz), iup(ixyz)
    n1=ind(i1,ixyz)
    if( irnk2(n1,iy) .ge. ilo(iy) .and.
        irnk2(n1,iy) .le. iup(iy) )then
      ilp = ilp +1
      index(ilp) = n1
    endif
  101 continue

c second intersection
  do 102 i1 = 1, ilp
    n1 = index(i1)
    if( irnk2(n1,iz) .ge. ilo(iz) .and.
        irnk2(n1,iz) .le. iup(iz) )then
      nlist = nlist + 1
      list(nlist) = n1
    endif
  102 continue
```

Figure 7. Indirect list intersection

The algorithm described in Figure 7 is an indirect list intersection. That terminology is used because of the indirect memory reference of the points in the `if` test: `irnk2(n1,iy)`, where `n1=ind(i1,ixyz)`. This implies `irnk2(ind(i1,ixyz),iy)` or an indirect memory reference. In order to directly reference the data, the following can be done, as shown in Figure 8.

```

do 10 i=1,npoints
  irnk(i,1,1) = irnk2(ind(i,1),2)
  irnk(i,1,2) = irnk2(ind(i,1),3)
  irnk(i,2,1) = irnk2(ind(i,2),1)
  irnk(i,2,2) = irnk2(ind(i,2),3)
  irnk(i,3,1) = irnk2(ind(i,3),1)
  irnk(i,3,2) = irnk2(ind(i,3),2)
12 continue

C first intersection
do 101 il = ilo(ixyz), iup(ixyz)
  if( irnk(il,ixyz,1) .ge. ilo(j,iy) .and.
  *   irnk(il,ixyz,1) .le. iup(j,iy)      )then
    ilp = ilp + 1
    index(ilp) = il
  ENDDIF
101 CONTINUE

c second intersection
do 102 il = 1, ilp
  if( irnk(index(il),ixyz,2) .ge. ilo(j,iz) .and.
  *   irnk(index(il),ixyz,2) .le. iup(j,iz)      )then
    nlist = nlist + 1
    list(nlist) = ind(index(il),ixyz)
  endif
102 continue

```

**Figure 8.** Direct list intersection

To understand how the movement of the indirect memory references out of the list intersection loops improves performance, consider the following data regarding central memory performance on the Cray Y-MP [3].

If no memory conflicts are encountered, the following access times for each register type can be expected:

- 19 plus vector length CPs (clock periods) for V register stride references
- 24 plus vector length CPs for V register gather references.

So to reference the `irnk2` array in the `do 101` loop in Figure 7 above would require  $19+ilp$  CPs for `ind(I1,ixyz)`, plus  $24+ntrip$  CPs for `irnk2(n1,iy)`, where  $ntrip = iup(ixyz) - ilo(ixyz) + 1$ . Recall that  $iup(ixyz) - ilo(ixyz) + 1$  is the length of the smallest list. This assessment assumes that the indirect memory references produce no delays due to memory conflicts. In reality, such a reference almost always produces some conflicts, so this assessment is generous. To continue, then, referencing `irnk2` indirectly costs  $43+2*ntrip$  CPs. Assuming that  $ntrip$  averages some fraction  $1/F$  of the number of points  $N$ , we have a time of  $43+2*N/F$  CPs.

The key point to the difference between indirect and direct list intersection is that this cost in time is going to be incurred for *every* list intersection operation that is performed. There are two list intersections in 3D for each box (search space) and in many applications the number of boxes is on the order of the number of points  $N$ . For the purposes of demonstration, the

number of boxes is assumed to be the same as the number of points. So, to do the indirect memory references for the whole search problem will involve a cost in time of  $43*N + 2*N^2/F$  CPs.

The optimized direct list intersection performs the indirect memory references on all of the rank arrays *only once*, prior to list intersection. In order to handle all possible cases, six scratch arrays are built to hold the results of the up-front indirect memory operations. This involves a cost in time of  $6*(43+2*N)$  CPs. Then, at list intersection time, the memory references are direct and **stride-1**. This is the most efficient kind of memory reference that can be done. It will involve a cost in time of  $19+N/F$  CPs for each box processed. So for the whole list intersection, the cost is  $19*N + N^2/F$  CPs. Adding the up-front gathers, the whole optimized search problem costs  $258 + 31*N + N^2/F$  CPs to process the **irnk2** array. Subtracting the cost for the optimized method from the cost for the original method, the savings is  $N^2/F + 12*N - 258$  CPs. For the CRAY Y-MP, one CP is 6 nanoseconds.

To see how the direct list intersection is an improvement, consider a set  $N=100000$  of randomly distributed points on the space  $0 < x,y,z < 1$ . Based on performance evaluation of the algorithm the value of  $F$  is approximately 20. Since 1 CP is 6 ns, the savings can be computed to be approximately 3 seconds. Factoring in the cost penalty of the original method due to memory conflicts is impossible to do because of the non-deterministic nature of the references, but given the fact that there is a difference of a factor of 5 in performance between the best memory access pattern and the worst [3], it is easy to see how the savings could be much more.

## 6 Performance

In this section the performance of the search algorithm is presented. With the realization that this algorithm may be used for several different applications, the performance is given for several point sets and box definitions. Three point sets are used, as shown in Figure 9. The first is a set of randomly distributed points on the space  $0 < x,y,z < 1$ . The second is a set of randomly distributed points in a diagonal rod with a radius of 0.1, beginning at  $x,y,z = 0,0,0$  and ending at  $x,y,z = 1,1,1$ . The third is a set of randomly distributed points in a diagonal rod impacting a plate.

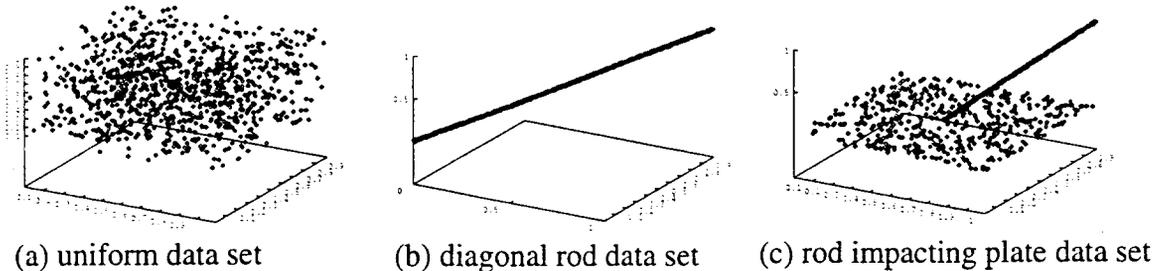


Figure 9. Three point sets used in performance testing

Two ways of covering the space  $0 < x,y,z < 1$  are used. The simplest is to uniformly cover the space by placing the boxes side by side - that is to have the boxes independent of the points. With this type of arrangement the number of boxes are assumed as:  $M = (\text{rint}(N^{1/3}))^3$ . The other way to cover the space  $0 < x,y,z < 1$  with boxes is to center a box around each point. The size of the box is then a free parameter and is usually related to a physical aspect of the problem. In the examples that are presented, the box size was chosen so that an average of ten particles are found inside each box. This case represents the likely scenario for particle methods.

The first step in the algorithm is the one-dimensional sorts of the points. In the following work, the sorting is done using a standard CRAY UNICOS<sup>®</sup> library routine `orders` [2]. The routine has a  $\text{Log}_2 N$  performance, as shown in Figure 10. The performance is independent of the geometric distribution of the point set.

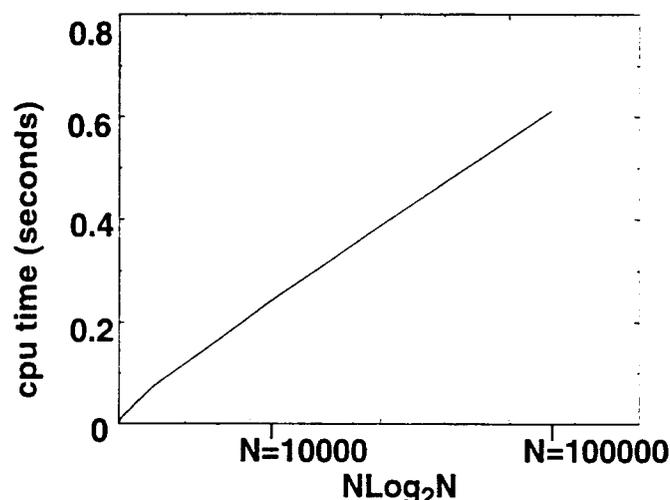


Figure 10. Timing for Cray UNICOS subroutine `orders`

The second step of the algorithm is the binary search. The performance of the classical (scalar) binary search, the  $\text{Log}_{64} N$  CRAY search, and the vectorized  $\text{Log}_2 N$  search are shown in Figure 11. Here, again, the performance is independent on the point set distribution.

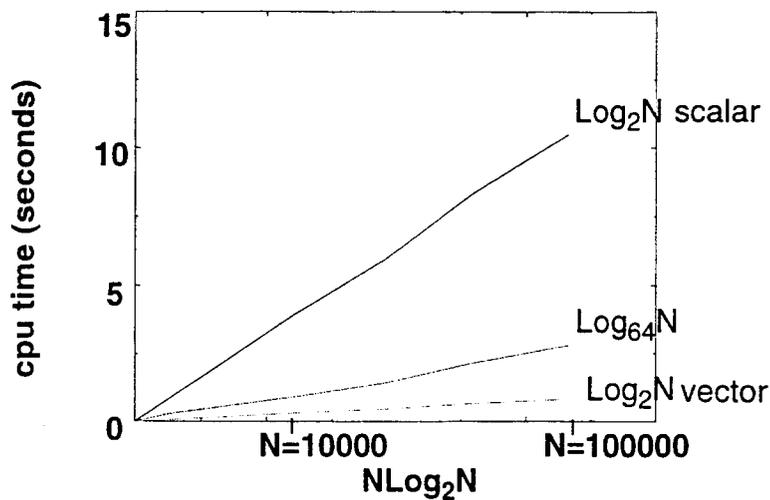


Figure 11. Timings for binary search

Finally, the performance for the complete algorithm is shown in Figures 12 and 13. The timings are dependent on the problem domain. The reason for this is that the number of points found in the smallest list varies with the point data set. It is important to realize, though, that the absolute worst case data set is uniformly distributed points over  $0 < x, y, z < 1$ .

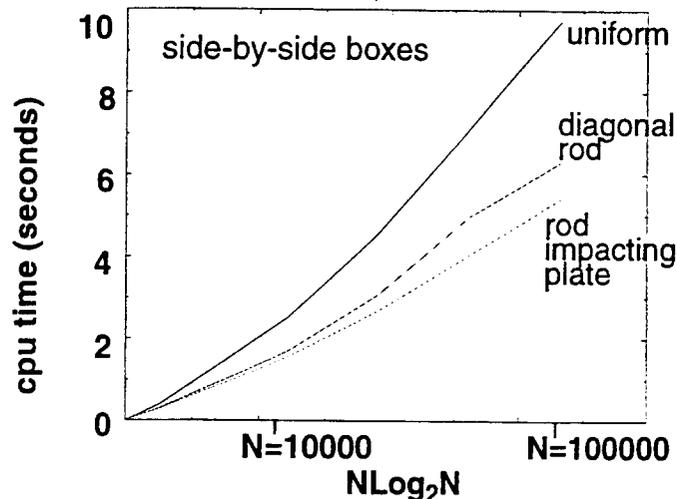


Figure 12. Timing for optimized point-in-box search algorithm: side-by-side boxes

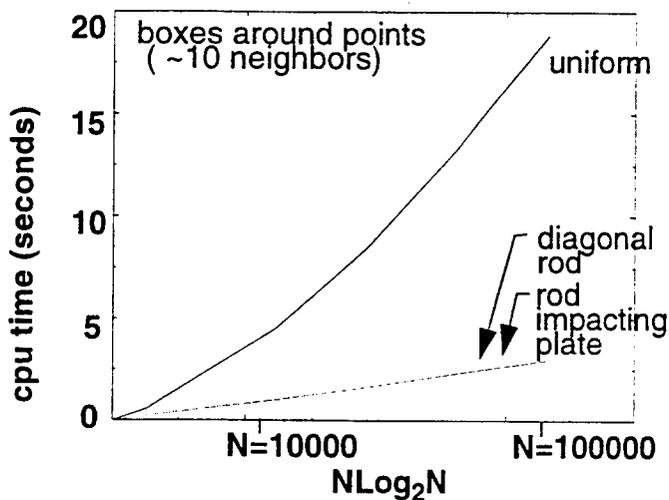
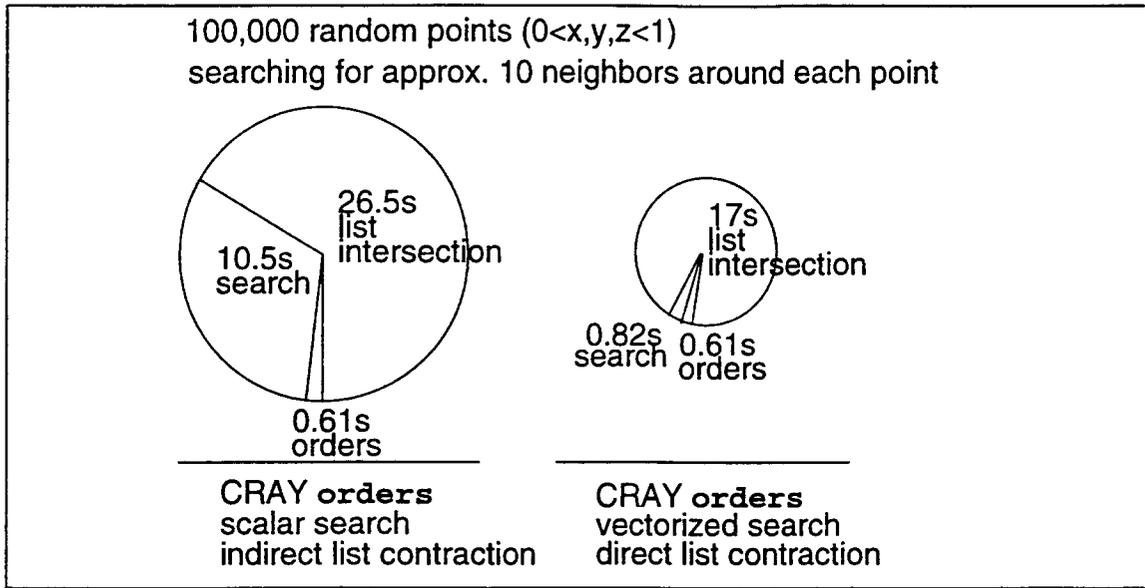


Figure 13. Timing for optimized point-in-box search algorithm: boxes around points

## 7 Summary

The point-in-box search algorithm developed by Swegle has been optimized by taking advantage of the architectural features of the CRAY YMP supercomputer. Figure 14 represents improvements in the performance of the algorithm for  $N=100000$  points uniformly distributed on the domain  $0 < x, y, z < 1$ . Recall that this point set represents the worst possible distribution of points for the performance of the algorithm. With some ordering of the points, such as a rod impacting a plate, the algorithm's performance is considerably better.



**Figure 14.** Performance improvement by optimizing the point-in-box search algorithm on the CRAY Y-MP

**REFERENCES**

[1] Swegle, J.W., "Search Algorithm," Internal Memorandum, Sandia National Laboratories, Albuquerque, NM 87185, May 25, 1992.

[2] UNICOS Math and Scientific Library Reference Manual, VOL. 3, August 1992.

[3] CRAY Y-MP System Programmer Reference Manual, June 1989.

## **Distribution**

MS0841	09100	Hommert, Paul J.
MS0443	09117	Morgan, Harold S.
MS0443	09117	Attaway, Steve W.. (10)
MS0807	04418	Davis, Mike E. (10)
MS0443	09117	Heinstein, Martin W. (10)
MS0443	09117	Swegle, Jeff W. (10)
1	MS 9018	Central Technical Files, 8940-2
2	MS 0899	Technical Library, 4916
1	MS 0619	Review & Approval Desk, 15102 For DOE/OSTI