15-853 Project: Parallel Callahan-Kosaraju Algorithm for All-Nearest-Neighbors

Aapo Kyrölä
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
akyrola@andrew.cmu.edu

Abstract

We implemented a parallel version of the Callahan-Kosarajun algorithm [1] for computing all-nearest-neighbors. Our C++/CILK implementation shows very good performance and decent parallel scalability. We also discuss challenges with parallel memory allocation which we had to overcome in order to gain any parallel speedup.

1 Introduction

For this project we implemented a parallel version of the Callahan-Kosaraju algorithm [1] (C-K algorithm) for efficiently computing *all-nearest-neighbors* for a large set of points. After a linear-time (in the number of points) precomputing step, C-K algorithm computes nearest-neighbor queries in constant time.

We will now give an informal introduction to the C-K algorithm. For a formal treatment, please see [1] or slides for the class. Actual code for the algorithm, and the parallelization, is discussed in the subsequent section. For the rest of the paper, we denote by S the point set, and with $n = |S|$ the number of points.

1.1 Step 1: Constructing k-d decomposition tree

The algorithm starts by creating a **k-d tree** of the point set S. K-d tree is built by recursively splitting points by the longest dimension of the bounding box enclosing the points. The procedure is presented visually in Figure 1. As a result algorithm returns a *binary tree*, with leaf nodes for each of the points. Depth of the tree depends on the distribution of the points; uniform distributions result in shallow trees. For each internal node, we record the associated bounding box. [1]

1.2 Step 2: Well-Separated Realization

We now describe the main innovation of the C-K algorithm. The idea is to create *interaction edges* between nodes in the decomposition tree, such that following requirements are fulfilled:

- From each leaf (point), there is a path to any other leaf that consists of tree edges up (zero or more), across an interaction edge and tree edges down.

[1] This procedure works well in practice, but for some datasets it might lead to an exponential running time. Callahan and Kosaraju present in [1] also a more robust procedure, which is based on storing the points in linked lists sorted by each dimension. Although the running time is better in a worst case, this algorithm is more difficult to parallelize. For this work, we therefore use the simpler algorithm.

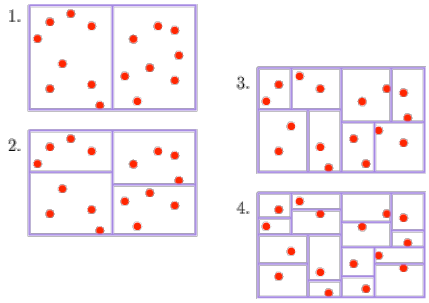

Figure 1: Computation of a k-d tree for a two dimensional pointset. Image credits: Guy Blelloch.

- Any nodes connected by an interaction are *well-separated*. See left side of the Figure 2 for a visual description.
- The number of interaction edges is $O(n)$.

Decomposition tree, supplied with the interaction edges is called *well-separated realization* (WSR). Algorithm for building the WSR is a triple-recursive algorithm, which we present in the next section. For the proof of the correctness, we refer reader to [1]. A simple well-separated realization is shown in Figure 2.

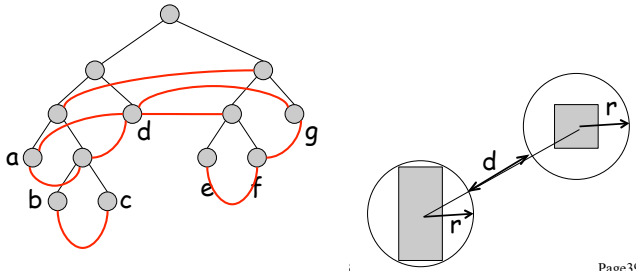

Figure 2: Left: Well-Separated Realization: A decomposition tree with interaction edges (red). Right: two set of points are well separated with factor s, if the distance d between circles enclosing them is larger than sr. r is the radius of the larger enclosing circle. For nearest neighbor algorithm, we use $s = 2$. Image credits: Guy Blelloch.

1.3 Step 3: Finding the nearest neighbor

Given the WSR graph, we can now find the nearest neighbor for a point p in constant time by following all the interaction edges from the associated leaf. It is easy to show that we do not need to follow interaction edges from parents of p, since any connected subset is *well-separated* from the parent and cannot thus contain a nearest-neighbor (i.e, any child of the parent would be closer).

The algorithm for finding K nearest neighbor is much more complicated, and we do not discuss it here.

2 Implementation

We implemented the algorithm in C++. For parallel primitives we used CILK+ [2]. Parallelization was straightforward in principle, but to achieve reasonable performance, we had to avoid frequent memory allocation. Our implementation was configured only for 3-dimensional pointsets, but is easily modified to support larger dimensions. Timings for the implementation are from experiments executed on a 8-CPU Intel Nehalem computer.

2.1 Data structures

We used an object-oriented data structure to present the tree and interaction edges. Each node is presented by objects of class `treenode`:

```
class treenode {
public:
  treenode * left;
  treenode * right;
  treenode * parent;
  box boundingbox;
  int point_idx;
  std::list<treenode *> * interacts;
  ...
    [methods]
};
```

Field `pointidx` is an index to an array of *point*-objects and is set only for leaf-nodes. Point-objects simply contain an array of coordinates.

Field `interacts` is a linked-list (using C++ STL list) of nodes.

2.2 Decomposition tree

The algorithm for building the k-d tree is a straightforward recursive algorithm; it resembles quicksort. For performance reasons, our implementation is very memory efficient and does not require memory allocation in the intermediate steps (see discussion below).

Algorithm takes as an argument a set of points P and a parent node. First it computes the bounding box for points in P and chooses the largest dimension d_m. New $node$ is created and the bounding box recorded. P is then divided into two point sets: first set contains the points having d_m coordinate less than the midpoint of the bounding box in that dimension; second set contains rest of the points. Algorithm is then called recursively for the two sub-pointsets, and resulting nodes attached as left and right node of `node`.

For the C++ implementation we avoid creating new arrays for the splitted point sets and simply pass pointers to the start and end of the sub-range of the point-array in question. The points are then sorted so that points in the first sub-pointset appear first. Thus, each call to the algorithm works on a fixed range of the point array and sorts points *in-place* inside its range. In this way, it is similar to a typical *quicksort* implementation in C/C++.

The sequential C++ implementation (with some details omitted) is below:

```
  treenode * btree(pointset_part p, treenode * parent, int level) {
    if (p.length == 1) {
      return leaf(parent, pointidx[p.begin]);
    } else {
      std::pair<int,double> dmax = maxpart_dim(p);
      // Split
      double midpoint = dmax.second;
      int end = p.begin + p.length;
      int c=p.begin;
      // Sort points according to whether they have less than
      //  or greater than position
      // in the largest dimension of the bounding box.
      for(int i=p.begin; i<end; i++) {
        if (points[pointidx[i]].coord[dmax.first] < midpoint) {
          workarray[c++] = pointidx[i];
        }
      }
      int p2start=c;
```

```
      for(int i=p.begin; i<end; i++) {
        if (points[pointidx[i]].coord[dmax.first] >= midpoint) {
          workarray[c++] = pointidx[i];
        }
      }
      // Move to correct parts of the array
      for(int i=p.begin; i<end; i++) {
        pointidx[i] = workarray[i];
      }
      treenode * node = newnode(parent, p2start);
      node->boundingbox = pointpart_box(p);
      node->lmax = lmax(node->boundingbox);
      node->left = btree(pointset_part(p.begin, p2start-p.begin),
       node, level+1);
      node->right = btree(pointset_part(p2start, c-p2start),
       node, level+1);
      return node;
    }
  }
```

2.2.1 Parallel version

Like with quicksort, we can use CILK to execute the recursive calls in parallel. As a simple optimization, we spawn parallel invocations only if we are high up in the recursion. We also only spawn the first recursive call in parallel.

```
     #ifdef CILK
        treenode * leftn;
        treenode * rightn;
        if (level < MAXPARALLELDEPTH) {
            leftn = cilk_spawn btree(pointset_part(p.begin,
             p2start-p.begin), node, level+1);
            rightn =   btree(pointset_part(p2start,
             c-p2start), node, level+1);
            cilk_sync;
        } else {
            rightn = btree(pointset_part(p2start, c-p2start),
             node, level+1);
            leftn = btree(pointset_part(p.begin, p2start-p.begin),
              node, level+1);
        }
        node->left = leftn;
        node->right = rightn;
      #else
       [sequential version]
```

2.2.2 Efficient memory allocation

Our initial tests showed no improvement in performance when run in parallel (on up to 8 cpus). Indeed, the tree building was sometimes even slower than sequentially. The culprit was memory allocation when creating new `treenode` -objects: a standard `malloc/new` is slow as it sequentializes memory allocations. But even CILK's more efficient memory manager ("miser") failed to scale, *miserably*. It seems that large number of small allocations are problematic for the memory manager. However, we did not investigate the issue further and it may be possible to configure *miser* to work better with our needs.

We solved the problem efficiently by preallocating all nodes. As the number of nodes in a binary tree is always $2n-1$, this does not waste any memory. We create two arrays of `treenode`-objects: one for internal nodes, `internalnodes` and another for leaf nodes, `leaves`. Leaves are placed to the array location in `leaves` corresponding to the index of the associated `point` in the point-

array. Internal nodes are placed into array `internalnodes` at index that is same as the index of the *splitting point*. That is, assume a call to `btree()` processes points that are in the point array at positions $[a..b]$. After choosing the dimension to split the points, it arranges points to split the range to two subranges $[a..c]$ and $[(c+1)..b]$. Then the new node will be placed in `internalnodes[c]`. It is easy to see that no internal nodes will get same array index.

Due to the preallocation, the tree-building algorithm does not need to allocate any memory, and there is no concurrent accesses to memory.

2.2.3 Parallel performance of the tree building

Following table displays some selected measurements for a sequential and 8-cpu execution of the tree building algorithm. Parallel speedup for large sets is $2.5x - 3x$. For smaller sets the speedup is smaller, as expected, due to overhead of CILK and multithreading.

Number of points	Tree depth	Sequential time (s)	Parallel - 8 cpus (s)	Speedup
4 K	16	0.0034	0.002	1.7
64 K	19	0.08	0.034	2.3
0.5 mil	23	0.9	0.35	2.5
2 mil	25	4.67	1.65	2.8
4.2 mil	27	10.3	3.6	2.86

2.3 Well-Separated Realization

Computing the well-separated realization (WSR) is a simple triple recursion that starts by call `wsr(root)`. CILK-parallelization is similar to the tree building algorithm. As a departure from the original algorithm, we add interaction edges only if other side of the edge is a leaf. Interactions between internal nodes are not relevant for the nearest neighbor computation.

```
void wsr(treenode * node) {
    if (node->left == NULL) {
      // is leaf
      return;
    } else {
      #ifdef CILK
        cilk_spawn wsr(node->left);
        wsr(node->right);
        cilk_sync;
      #else
        wsr(node->left);
        wsr(node->right);
      #endif
      wsrPair(node->left, node->right);
    }
  }

  void wsrPair(treenode * t1, treenode * t2) {
    if (wellSeparated(t1->boundingbox, t2->boundingbox)) {
     // Add interactions only to leaves.
      if (t1->left == NULL) t1->add_interaction(t2);
      if (t2->left == NULL) t2->add_interaction(t1);
    } else if ((t1->lmax) > (t2->lmax)) {
      wsrPair(t1->left, t2);
      wsrPair(t1->right, t2);
    } else {
      wsrPair(t1, t2->left);
      wsrPair(t1, t2->right);
    }
  }
```

2.3.1 Parallel performance of the WSR computation

Following table lists selected performance measurements for the WSR computation. Parallel scalability is similar to the tree building.

Number of points	Tree depth	Sequential time (s)	Parallel - 8 cpus (s)	Speedup
4 K	16	0.011	0.007	1.6
64 K	19	0.27	0.14	1.9
0.5 mil	23	2.4	1.02	2.3
2 mil	25	9.8	3.96	2.5
4.2 mil	27	19.8	7.6	2.6

2.4 Nearest Neighbor given WSR

Finally, we need to compute nearest neighbor for each point. This takes only constant time per point, since we need to only follow interaction edges from each leaf, not from internal nodes. Parallelization is simply done by computing nearest neighbors for several points in parallel.

```
std::pair<int,double> mindist(point & p, treenode * node, int minDistIdx, double mi
   if (node->left == NULL) {
     double r2 = p.rsqr(points[node->point_idx]);
     if (r2<minDistSqr) return std::pair<int,double>(node->point_idx, r2);
     else return std::pair<int,double>(minDistIdx, minDistSqr);
   } else {
     std::pair<int,double> minleft = mindist(p, node->left, minDistIdx, minDistSqr)
     return mindist(p, node->right, minleft.first, minleft.second);
   }
 }


 void compute_wsr_nn() {
   int psz = points.size();
 #ifdef CILK
    cilk_for(int pntidx=0; pntidx<psz; pntidx++){
 #else
    for(int pntidx=0; pntidx<psz; pntidx++){
 #endif
      double minDistSqr = 1e30;
      int minDistIdx = -1;
      point & p = points[pntidx];
      std::list<treenode *> & interacts = *(p.node->interacts);
      std::list<treenode *>::iterator iter;
      for(iter = interacts.begin(); iter != interacts.end(); iter++) {
        std::pair<int, double> mind = mindist(p, *iter, minDistIdx, minDistSqr);
        minDistIdx = mind.first;
        minDistSqr = mind.second;
      }
      p.nn = minDistIdx;
   }
 }
```

2.4.1 Parallel performance of the nearest neighbor

Following table shows the parallel performance of computing nearest neighbors for points in parallel (given precomputed WSR). The computation is embarrassingly parallel, and we experience good speedup. However, the speedup on larger point sets is still short from the perfect $8x$ speedup, significantly. This is probably due to large memory footprint which satures the memory bus. However, we did not analyze this further.

Number of points	Tree depth	Sequential time (s)	Parallel - 8 cpus (s)	Speedup
4 K	16	0.0048	0.0014	3.4
64 K	19	0.30	0.054	5.6
0.5 mil	23	3.61	0.68	5.3
2 mil	25	17.2	3.2	5.4
4.2 mil	27	38.1	7.1	5.4

3 Experiments

3.1 Setup

We run experiments with three types of three-dimensional point sets. *Grid* -sets were simple two-dimensional equally spaced grids of points. Spatial decomposition works optimally with such distribution. *Random* is a randomly distributed set of points. Same seed was used for each run. *Shell* is a shell-type point cloud (point generation code borrowed from NVIDIA's nbody-demo for GPUs). See Figure 3 for examples of these pointsets.

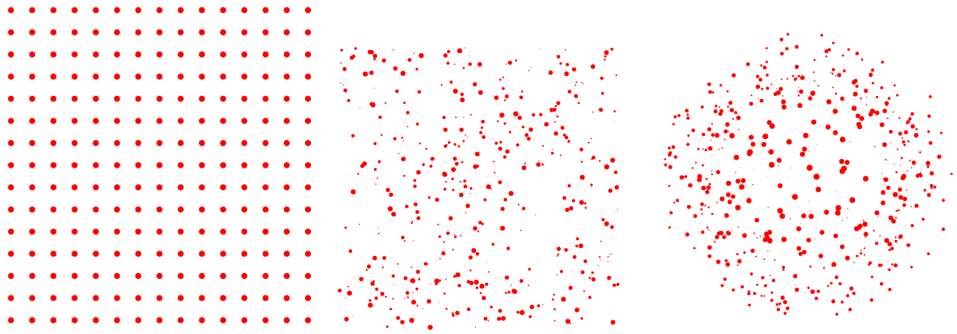

Figure 3: From the left: grid, random and shell -form set of points. Bigger points are closer to "camera" than smaller. Different distribution entails different structure of the spatial decomposition tree.

We run the algorithm on 1 and 8 cpus for several different sizes of point sets, ranging from hundreds to millions of points. All quoted timings include the precomputation steps, and the actual nearest neighbor computation, but not time used for creating the point cloud or validating the results. We validated C-K algorithm's nearest neighbor computation by computing neighbors with the *naive* nearest neighbor algorithm for a subset of points. Naive algorithm simply enumerates, for each point, all other points to find the closest, and thus runs in $O(n^2)$ time. C-K algorithm was able to outperform naive algorithm in all but very small datasets.

Figure 4 shows the number of interaction edges for each type of point set as a function of the number of points. It is evident that Grid is very different from the two other types.

3.2 Scalability with the size of pointset

According to the theory, C-K should scale linearly with the number of points. However, also the distribution of the points has an effect of the running time. As Figure 5 shows, the algorithm really scales linearly, when the underlying structure of the point set does not change.

3.3 Parallel performance

In previous section we measured the parallel performance of individual steps of the algorithm. Figure 6 shows the total parallel speedup of the algorithm. We compiled a separate version of the program for sequential runs to avoid any CILK overhead. On large datasets, the speedup is close to $4x$ on random and shell point sets, but with grid the scalability is clearly poorer. This is likely due to two reasons: (1) shallow decomposition tree, of whose construction provides less opportunity

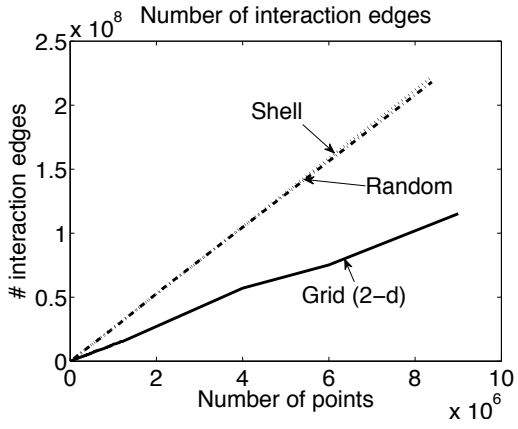

Figure 4: Number of interaction edges (size of WRS), as a function of pointset size.

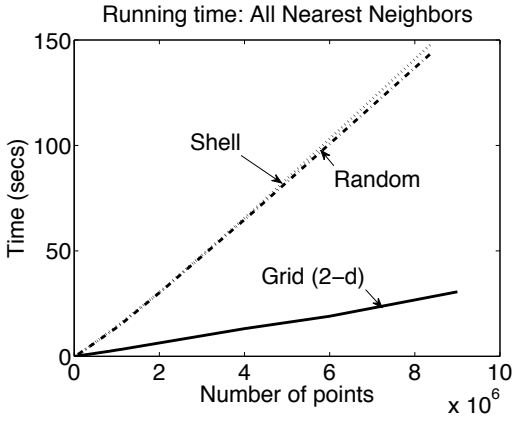

Figure 5: Running time as function of pointset size.

for parallelism, (2) point set has only two effective dimensions, which makes distance computations faster than with three dimensions. Less computation per memory access puts relatively more pressure on memory bus. However, we did not study this in detail.

3.4 Speedup over Naive Algorithm

And last, Figure 7 shows the dramatic improvement over the naive all-nearest-neighbors algorithm. The naive algorithm scales almost perfectly in parallel and thus is relatively less slower when run in parallel. Already with point sets of mere thousands of points, the speedup is very significant.

Remark: the naive running time is estimated from the time taken to compute nearest neighbors for a small sample of points.

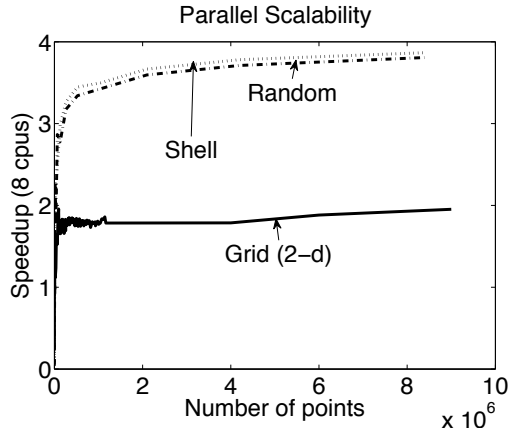

Figure 6: Speedup (sequential running time / parallel running time) on a 8-core Intel Nehalem, as a function of pointset size.

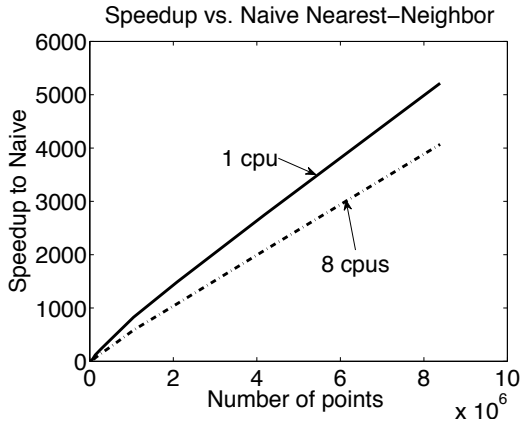

Figure 7: Speedup over naive algorithm.

References

[1] P.B. Callahan and S.R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM (JACM)*, 42(1):67–90, 1995.

[2] I. Cilk+. Software Development Kit. *Intel Corporation, Feb*, 2010.