Parallel programming project report

Varun Agarwal —— Aditya Tomer
111491232, 111491409
atomer , avarun @cs.stonybrook.edu

***Abstract*—We implemented Callahan-Kosaraju algorithm for computing the n-nearest neighbours for n-points in a sequential and parallel manner. We used C++/CILK for our parallel implementation, which shows a high degree of parallelism for the algorithm. We also discussed problems faced by us while improving parallel performance and steps taken to achieve higher degree of parallelism. Paper consists of various graphical visualization helping us to understand the result achieved.**

I. Problem Description

NEAREST neighbour search (NNS), is a form of proximity search. It is optimization of a problem of finding a point closest(maybe similar) to a any other point in a set S. The measure of closeness is expressed in terms of dissimilarity function i.e. the less similar are two objects, less is the closeness between them. Formally, the nearest-neighbor (NN) search can be defined as: given a set S of points in a space M and a query point q $\in$ M , find the closest point in S to q . A direct generalization of this problem is a k-NN search, where we need to find the k closest points. We are trying to solve the problem where k=N. Where we have N data-points and wish to know which is the nearest neighbor for every one of those N points . Algorithm can be acronymed as All-Nearest-Neighbours.
The key application areas of All-Nearest-Neighbours is as follows -

- Approximation of interaction forces on each of the N bodies due to remaining N-1 bodies.
- Approximation of entropy of the system.
- Used in astronomy to simulate the motion of stars and other mass.
- Used in biology to simulate protein folding mechanism.
- Used in Engineering to simulate PDEs (can be better than Finite Element Meshes for certain problems)
- Used in machine learning to calculate certain Kernels and makes predictions using the training dataset directly.
- Approximation of the force/potential due to a set of points by a multipole expansion truncated to a fixed number of terms (sort of like a taylor series).

Faculty: Prof. Rezaul Chowdhary

The naive algorithm of finding the N nearest neighbour is O(N^2). Using Kallahan-Kosaraju we can get approximate answer in O(n) time plus the time required to build the tree. After parallelizing Kallahan-Kosaraju, we can calculate N nearest neighbour in constant time, after a linear time (in the number of points) precompute step.

II. Prior Work

In pattern recognition, the k-nearest neighbors algorithm (k-NN) is a non-parametric method used for classification and regression.[1] In both cases, the input consists of the k closest training examples in the feature space. The output depends on whether k-NN is used for classification or regression: k-NN is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until classification. The k-NN algorithm is among the simplest of all machine learning algorithms.
The nearest neighbour algorithm was one of the first algorithms used to determine a solution to the travelling salesman problem. In it, the salesman starts at a random city and repeatedly visits the nearest city until all have been visited. It quickly yields a short tour, but usually not the optimal one.

A Parallel Implementation of the K Nearest Neighbours Classifier in Three Levels: Threads, MPI Processes and the Grid by G. Aparcio, I. Blanquer, V. Hernandez
A CUDA-based parallel implementation of K-nearest neighbor algorithm published in Cyber-Enabled Distributed Computing and Knowledge Discovery, 2009. CyberC '09. International Conference
one of implementation used Neural Networks for constructing a multi-layer feed-forward network that implements exactly a 1-NN rule. The advantage of this approach is that the resulting network can be implemented efficiently. The disadvantage is that the training time cant grow exponentially for high dimensional pattern spaces, which could make it impractical
Aapo Kyrl[1] have implemented the parallel version of the Kallahan-Kosaraju all-nearest neighbour algorithm using C++, Cilk+. The paper shows a good parallel performance and decent parallel scalability of the algorithm, also discussing challenges faced during parallelizing. The paper shows considerable improvements over naive algorithm, and scales perfectly for large number of points but performs slower for very small dataset. In this project we propose to analyze each of these paper findings, plot graphs for various improvements over naive and linear Kallahan-Kosaraju algorithm.

A. Goal(s)/Deliverable(s)

- Implementation of sequential Kallahan-Kosaraju for finding the nearest neighbors.
  - This particular goal has been divided further into subgoals
  - Develop the sequential algorithm to construct the K-d tree -
    - Construct K-D tree by defining the bounding box that encloses all the points
    - Determine the longest dimension of the bounding box
    - Recursively split points by that longest dimension
    - As a result algorithm returns a binary tree, with leaf nodes for each of the points.
    - Process is repeated for recursively generated smaller dimensions.
  - Create interaction edges between nodes in the decomposition tree
  - Develop the algorithm for computing k-nearest neighbours.
  - Generalize the algorithm for N-nearest neighbours.
- Implementation of Parallel Callahan-Kosaraju algorithm using C/C++ Cilk+
  - Constructing k-d decomposition tree, evaluating parallel performance of the k-d tree building.
  - The idea is to create interaction edges between nodes in the decomposition tree
  - Decomposition tree, supplied with the interaction edges is called well-separated realization (WSR), Compute the well-separated realization (WSR)
  - Based on WSR and interaction-edges we need to find the nearest neighbour for each point.
- Efficient memory allocation precompute step to improve performance of parallel algorithm.
  - Currently we are not following this step, we will take this performance improvement task towards the end of the project.
- Plotting visualization in tabular and graphical format for different dataset sizes. .
- Implementation of the Parallel Kallahan-Kosaraju algorithm using GPUs for K-nearest neighbours.

B. Brief Overview

The steps for algorithm are -
1. We implemented a points generator which can generate points in 3D-random, grid and spherical manner.
2. Once the points are generated, preallocation of memory for KD Tree is done. The leaf of the KD tree represents total points. Hence total nodes including internal and leaf nodes is 2n-1.
3. Memory is preallocated for leaf as well as internal nodes separately in different pointer array.
4. KD Tree is now created for all the points using the bounding box technique.
5. Once the KD Tree is created, the root of the tree is used to create interaction edges between the leaf nodes(points) to the other points. Only those interaction edges are created which are well separated from each other such that d ¿ r.s
6. Now all points in the system have their interaction edges created. In Parallel for all the points interaction edges are visited and minimum distances between each of the points is recored in.

III. Implementation

A. Generating Input

Generating Points
The algorithm starts by genearting sequence of points in either random, grid or spherical manner. Once the points are generated.

Pre Memory Allocation
Once the points are generated, we did the preallocation of memory for the KD Tree. The total leaf of the KD Tree is same as that of the total number of points. Since the KD tree is a binary tree with n leaf nodes, the total nodes in the tree will be 2n-1. Where n is the number of points. 2 pointer arrays are created one for each internal and leaf nodes of the tree. Preallocation of memory helped us in achieving higher parallellism as compared to allocation of memory while building the tree. In general the standard new or malloc allocation is sequential memory allocation and it resulted in a hit to acheiving higher parallelism. Each of the preallocated nodes are placed at splitting index of the tree in the array. Example is if the kd tree is processing the nodes from [min,max] than the splitting will be done at the mid point (min+max)/2. Pre allocation step helped to achieve higher concurrency.

KD Tree construction
Now the KD Tree construction is done. KD Tree is recursively built by splitting the longest dimension of the bounding box enclosing the points. The algorithm need the parent node and the set of points for that node on which it needs to act. So firstly, the algorithm creates a virtual box for all the points it needs to enclose. Once it is done the longest dimension across all is found. Now all the points are splitted based on if they are less than the mid point or greater than the mid point in that dimension. As a result, a binary tree is constructed with leaf nodes for the points. Here parallelism is achieved as both the branches of the tree during tree construction can be done in a parallel manner so a new thread is created using cilk_spawn for left branch and the main thread itself constructs the right branch. Depth of the tree is based on randomness/uniformity of the tree. The sequential algorithm here runs in $\mathbf{O(nlog(n))}$. However using parallelism this step can be done in log(n) time.

Well Separated Region This is a tripple recursive algorithm starts by calling root of the kd tree. The idea is to create interaction edges between the leaf nodes(points) to

other points which are well separated as given by section well-separated. Since the internal nodes represents a set of points, it is irrelevant to consider them for nearest neighbour for a point. The sequential algorithm will take **O(nlog(n))** time but with parallelism the time will reduce to **log(n)**.

Computing N-Neighbour for points Finally we need to compute the n-neighbour for each of the points. It takes constant time per point for a nearest-neighbour calculation by following the interaction edges. Parallelism is simple done by calculating the nearest-neighbour for all the points in parallel. This bring the algorithm running time from **O(n)** to **O(log(n))**.

Work= O(nlog(n)) + O(nlog(n) + O(n)
Span = O($log^2(n)$+$log^2(n)$)+O(log(n))
Parallesim = O($n/log(n)$)

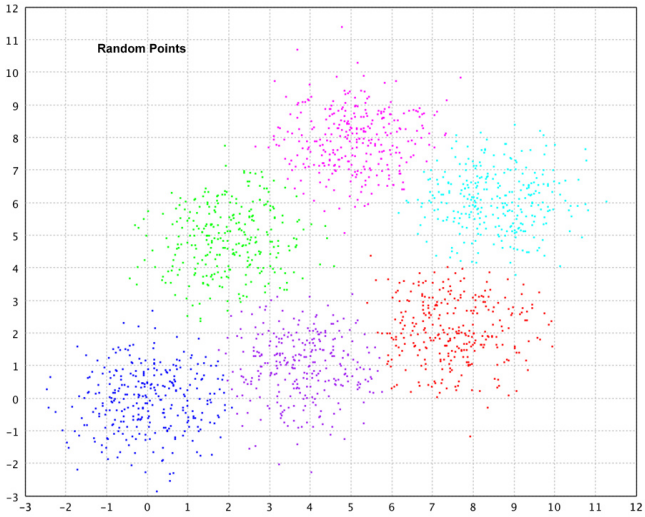


Fig. 1. points existing in random space

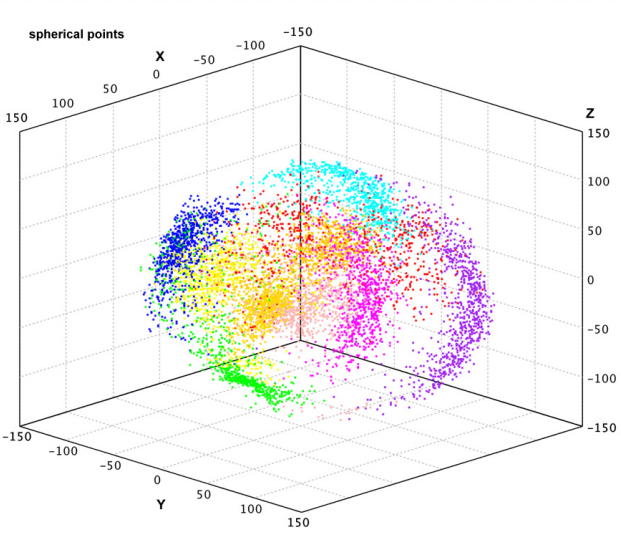
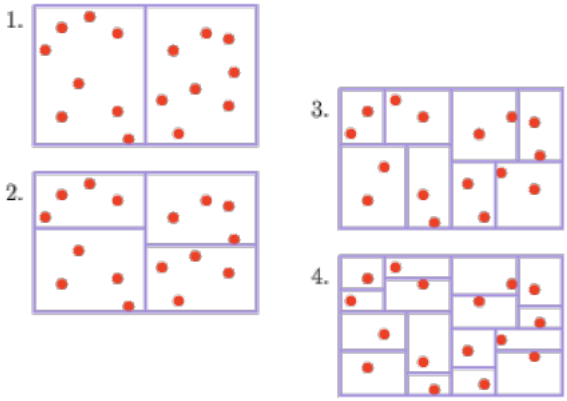


Fig. 2. points existing in spherical space

dimension of the bounding box enclosing the points. 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.



B. Constructing k-d decomposition tree

The algorithm starts by creating a k-d tree of the point set . K-d tree is built by recursively splitting points by the longest

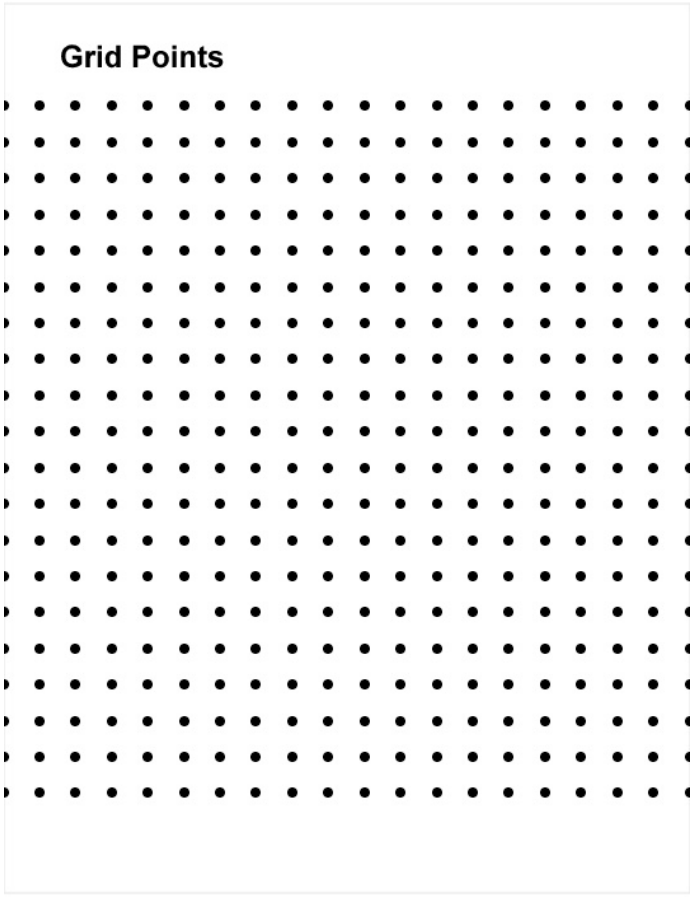
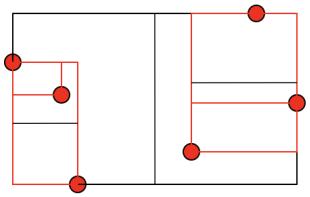


Fig. 3. points existing in grid space

Function Tree(P)
if |P| = 1 then **return** leaf(P)
else
d_{max} = dimension of l_{max}
P_1, P_2 = split P along d_{max} at midpoint
Return Node(Tree(P_1),Tree(P_2), l_{max})



C. *interaction edges in the decomposition tree*

The idea is to create interaction edges between nodes in the decomposition tree, such that following requirements are fulfilled:

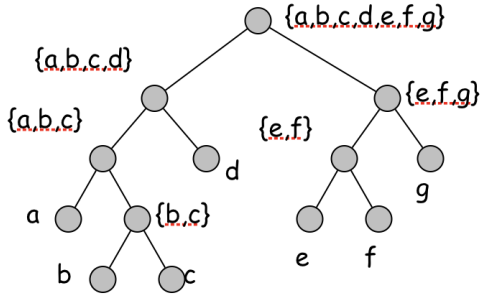
1. From each leaf (point), there is a path to any other leaf that consists of tree edges up (zero or more), across an interaction edge and tree edges down.

2.Any nodes connected by an interaction are well-separated. See left side of the Figure 2 for a visual description.

3.The number of interaction edges is O(n).

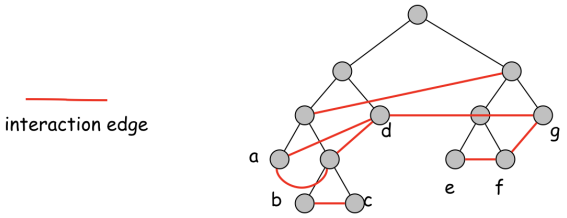
Decomposition tree, supplied with the interaction edges is called well-separated realization (WSR). Algorithm for building the WSR is a triple-recursive algorithm, which we present in the next section. For the proof of the correctness, we refer reader to [1]. A simple well-separated realization is shown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

A spatial decomposition of points



Realization

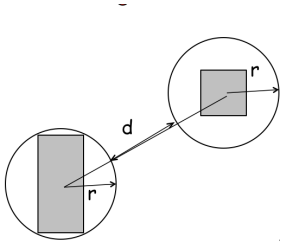
A single path between any two leaves consisting of tree edges up, an interaction edge across, and tree edges down.



A decomposition tree with interaction edges (red).
What is a well separated realization?

A realization such that the endpoints of each interaction edge is well separated

Well Separated:
r = smallest radius that can contain both rectangles
s = separation constant
d > s * r



15-853

Fig. 4. Example of a parametric plot $(\sin(x), \cos(x), x)$

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.

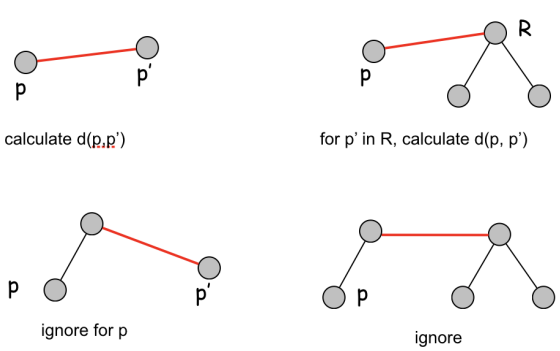
Overall approach comes out to be something like this. We Build tree decomposition: O(n log n) time and a well-separated realization: O(n) time. The Depth of tree is in O(n) worst case scenario, Hence we can bound number of interaction edges to O(n) so for both n-body and nearest-neighbors we only need to look at the interaction edges

D. Computing 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).

To find the nearest neighbor lets follow some steps:-
1. Build well-separated pair decomposition with s = 2
2. Recall that d $\geq$ sr = 2r to be well separated
3. The furthest any pair of points can be to each other within one of the rectangles is 2r
Therefore if d ¿ 2r then for a point in R_1 there must be another point in R_1 that is as close as any point in R_2. Therefore we dont need to consider any points in R_2.
Now to find nearest neighbor of every point:-
lets consider a point p. It interacts with all other points p through an interaction edge that goes from:

- p to $p^{'}$
- p to an ancestor R of $p^{'}$
- an ancestor of p to $p^{'}$ or ancestor of $p^{'}$



We take the minimum of all the distances calculated.

Calculate the forces among n bodys. Nave method requires considering all pairs and takes O(n^2) time. Using Kallahan-Kosaraju can get approximate answer in O(n) time plus the time to build the tree.

IV. RESULTS

To test our experiments, we had access to stampede supercomputer. It has 68 compute nodes with NVIDIA Kepler K20 GPUs2. to have our program up and running we followed some steps

- git clone $https://github.com/adityatomer/n\_nearest\_neighbour\_parallel.git$
- cd $WORK/Group19\_Aditya\_111491409\_Varun\_111491232\_Project$ on Stampede2
- icc -std=c++11 main.cpp -o out
- ./out 10000 4

A. Machine Configuration

Stampede2 hosts 4,200 KNL compute nodes, including 504 KNL nodes that were formerly configured as a Stampede1 sub-system.
Each of Stampede2's KNL nodes includes 96GB of traditional DDR4 Random Access Memory (RAM). They also feature an additional 16GB of high bandwidth, on-package memory known as Multi-Channel Dynamic Random Access Memory (MCDRAM) that is up to four times faster than DDR4. The KNL's memory is configurable in two important ways: there are BIOS settings that determine at boot time the processor's memory mode and cluster mode. The processor's memory mode determines whether the fast MCDRAM operates as RAM, as direct-mapped L3 cache, or as a mixture of the two. The cluster mode determines the mechanisms for achieving cache coherency, which in turn determines latency: roughly speaking, this mode specifies the degree to which some memory addresses are "closer" to some cores than to others. See "Programming and Performance: KNL" below for a top-level description of these and other available memory and cluster modes

Below figures shows parallelism graph for varied number of points for different processors for random, spherical or grid points.

In Figure 5 and 6 we have plot of all three input format against running time when ran for 1000 to 1000000 points. Now We observe that random and shell almost behave similar. However finding nearest neighbor is much faster when it comes to grid. Possible explanation for this can be that points are properly arranges and you have a fair idea of existence of another point.

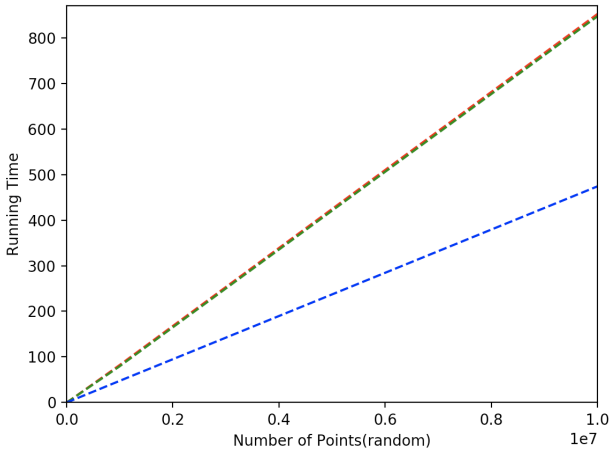


Fig. 5. Plot of points in all three input format vs running time with 1 processor. Shell and Random point graph are overlapped whereas the grid point graph can be easily seen.

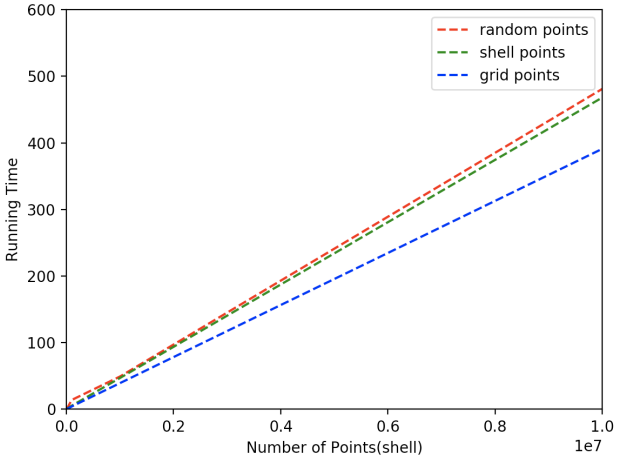


Fig. 6. Plot of points in all three input format vs running time with 1 processor.

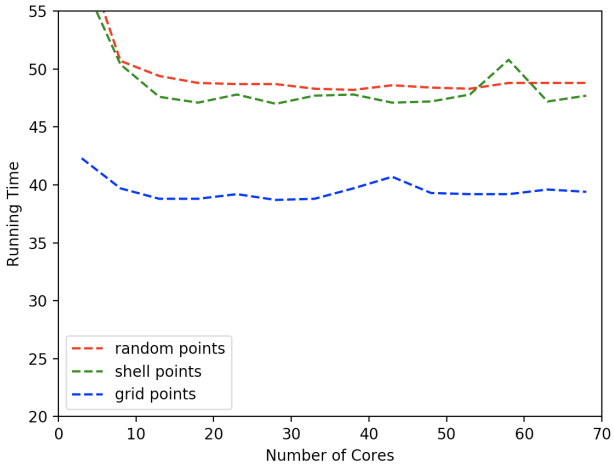


Fig. 7. input size = 1000000 points, number of cores varied

V. Extending the work

References

[1] https://en.wikipedia.org/wiki/Nearest_neighbor_search
[2] https://www.cs.cmu.edu/afs/cs/project/pscico-guyb/realworld/www/slidesF09/nn1.pdf
[3] https://www.coursera.org/learn/ml-clustering-and-retrieval/lecture/BkZTg/complexityof-nn-search-with-kd-trees
[4] https://www.cs.cmu.edu/~akyrola/files/kyrola15853CallahanKosaraju.pdf
[5] 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):6790, 1995.
[6] I. Cilk+. Software Development Kit. Intel Corporation, Feb, 2010. J. Hagenauer, E. Offer, and L. Papke. Iterative decoding of binary block and convolutional codes. *IEEE Trans. Inform. Theory*, vol. 42, no. 2, pp. 429-445, Mar. 1996.
[7] https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm