A well-separated pairs decomposition algorithm for k-d trees implemented on multi-core architectures

A well-separated pairs decomposition algorithm for k-d trees implemented on multi-core architectures

Raul H. C. Lopes[1]**, Ivan D. Reid**[2]**, Peter R. Hobson**[3]

[1,2,3]Particle Physics Group, Brunel University, Uxbridge, UB8 3PH, UK

E-mail:
[1]`raul.lopes@brunel.ac.uk`,[2]`ivan.reid@brunel.ac.uk`,[3]`peter.hobson@brunel.ac.uk`

Abstract. Variations of k-d trees represent a fundamental data structure used in Computational Geometry with numerous applications in science. For example particle track fitting in the software of the LHC experiments, and in simulations of N-body systems in the study of dynamics of interacting galaxies, particle beam physics, and molecular dynamics in biochemistry. The many-body tree methods devised by Barnes and Hutt in the 1980s and the Fast Multipole Method introduced in 1987 by Greengard and Rokhlin use variants of k-d trees to reduce the computation time upper bounds to $O(n \log n)$ and even $O(n)$ from $O(n^2)$. We present an algorithm that uses the principle of well-separated pairs decomposition to always produce compressed trees in $O(n \log n)$ work. We present and evaluate parallel implementations for the algorithm that can take advantage of multi-core architectures.

1. Introduction

Given a set P of n points in R^d, consider the following problems of: finding the two closest points to each other belonging to P; for each $q \in P$, finding its closest neighbour in $P - q$; and finding all k nearest neighbours of each $q \in P$.

Efficient Algorithms to solve these three problems are fundamental in areas like Computational Geometry, Machine Learning, or simulations and multivariate analysis in High Energy Physics, for example. They can all be solved sequentially within $O(n \log n)$ work ([1] and [2]). In this paper, we deal with how to solve them in parallel architectures, preserving the work boundary, but with algorithms that scale with the number of available processors.

In particular, the classical solution for the closest pair problem can throw light into the approach taken. The problem can be solved in $O(dn^2)$ time by computing all distances between all pairs of points and selecting the smallest. The standard textbook [3] solution when $d = 2$ consists in partitioning the initial set P into subsets P_L and P_R, and then computing:

- the closest pair for P_L;
- the closest pair for P_R;
- the closest pair (p, q) under the following restrictions: $p \in P_L$, and $q \in P_R$, and both p and q must lie within distance δ of the line separating P_L from P_R, where δ is the minimum of the distances for the pairs computed in the previous steps.

The closest pair is then selected from the three pairs computed above.

Essential to this solution is the divide-and-conquer nature of the process which leads to a $O(n \log n)$ time when the initial partition gives sets P_L and P_R with roughly the same number

of points. In addition, the recursion implied builds a tree that when explicitly constructed can be used to provide a straightforward and efficient solution for the problem of finding the closest neighbour to each point $q \in P$, achieved by a procedure that for each q computes in $O(\log n)$ its nearest neighbour (the nearest port-office problem in [4]).

Vaidya [5] first showed that if the set of all pairs of subsets found in a tree structure as mentioned above is computed, then the problem of finding all k nearest neighbours for all points in P can also be solved within $O(n \log n)$ time.

The main target of this paper is to describe parallel algorithms that can be used to build the tree structure implicit in the recursion implicit in the solution of the closest pair above. We will show performance results for the implementation on multi-core machines.

2. K-ary trees

Organising multidimensional point data is a central issue in computer graphics, image processing, computer geometry, to name just a few. A point in this context is a meaningful set of attributes from possibly different domains. Organizing means finding an effective representation that can help with solving efficiently, for example, search questions like those introduced in section 1. The structures involved in such organisation lead invariably to k-ary trees and search, k being equal to 2^d, where d is the number of attributes in each point, i.e., the number of dimensions of the space considered.

In the context of organising points in 2 dimensions, Finkel and Bentley [6] introduced the quad tree. A quad tree applies the idea of divide-and-conquer to each of the two attributes of a point. It imposes a grid-like structure on the region of space defined by the two-dimensional collection of the points in question. It is a tree structure where each node identifies one of the points as a discriminator to partition a set of points in exactly four quadrants. In this sense, it applies a parallel comparison over all, in this case two, attributes of the points.

Samet [1] identifies this property of comparing all attributes of each point as an advantage in a multiprocessing environment. On the other hand, this same property is at root of the difficulties in using them. First, a quad tree node can have up to three null descendants, as opposed to a binary that would have only one. This adds space cost. Also, a quad tree node uses more space than a binary tree node. In addition, extensions of quad trees to three or more dimensions add to these costs due to what Moore in [7] identified as the curse of dimensionality, a fast increasing number of empty regions (descendant nodes) growing with the dimensionality. Last, the recursive nature inherent to any tree structure becomes more complicated when the number of dimensions increases and creates serious difficulties with splitting and balancing work among processors in the case of parallel processing.

Introduced by Bentley in [8], k-d trees use a representation that very much mirrors that of a binary tree. Each node partitions a set of points on one attribute. On a path from root to leaf, the nodes cycle through the attributes (dimensions) used to partition the set of points. Associated with a node is a dimension and a point of partition. In Bentley's proposal for both the quad tree and the k-d tree, the point used for partitioning comes from the set being partitioned. An immediate advantage of the k-d tree is that simplicity of its nodes makes it suitable for ease organisation of points in more than two dimensions. Also unbalancing and waste of space due to empty regions in partitions could be less accentuated than with quad trees. Potentially, the sequential nature of comparison of attributes for one given point might be a hindrance for parallel construction, as pointed out by Samet [1].

Cycling through the dimensions, as proposed by Bentley, can lead to unnecessarily unbalanced trees with consequent waste in space and time in the construction and search procedures. Moore [7] and Omohundro [9] show extensive quantitative studies of k-d trees to identify situations that can lead to unbalanced trees. In the next section, a variation of k-d trees is introduced to exploit parallelism in construction while keeping the height of the trees inside $O(\log n)$.

3. Fair split and k-d trees

In this section and the ones that follow, k-d trees are the data structure considered for organising multidimensional data. A k-d tree is viewed essentially as binary search tree, with one added discriminator per node, which identifies the dimension over which the node partitions the region of space being considered.

A k-d tree can be stated by an inductive definition.

Definition 1 *A k-d tree over τ^d is an inductive type which is either*

- *a* leaf *containing just one element (point) of type τ;*
- *or it is a* node *composed of*
    - *a* discriminator*, which defines the dimension of split;*
    - *a* cut value *of type τ, which defines the value used to to split;*
    - *a* left *k-d tree over τ;*
    - *a* right *k-d tree over τ.*

Given a tree t following such inductive definition, it is required that two invariants hold:

- Any point in its *left* tree must have a value on the dimension defined by the *discriminator* of t less than the *cut value* of t.
- Any point in its *right* tree must have a value on the dimension defined by the *discriminator* of t greater than the *cut value* of t.

This is essentially Bentley's definition of a k-d tree. However, two important alterations are introduced:

- A node does not necessarily use a point of a set being partitioned as a split point. Any value will do, if it partitions the set in *balanced* way.
- Bentley recommends that the *discriminators* used in each a node should cycle through the dimensions of the set being indexed. In this work, the *discriminator* used in each node is *chosen* so that it will produce a balanced split.

The following considerations should be taken into account to build the trees following the definition above:

- The height of the final tree should be kept within $O(\log n)$, when the initial set has n points.
- The splitting procedure used to construct each node and the recursive procedure used to build the tree should always maximise the usage of the parallel computational resources available.

Vaidya in [5] and Callahan in [10] use the idea of splitting to produce a *well separated pairs* of subsets of initial set of points. Callahan introduces a concept of a fair split. The set of points being partitioned by a node defines a range in the dimension of the split. The cut value of split should divide the range close to half.

Callahan introduces both sequential and parallel k-d trees. The sequential algorithm is based on what Bentley called "burning the candle from both ends" in his algorithm for in situ partition of the Quicksort [11]. Callahan's sequential algorithm works with sorted versions of the set of points: one sorted version for each dimension. The algorithm searches for a split from left to right and right to left. The algorithm also adds intermediate sequences to add the split and pointers from and to the original sequence.

Callahan's method is essentially sequential. To add parallelism, Callahan adds a step where processors must guess *candidate slabs* for partitioning the set in each node, a process not easily realizable in practice and whose correctness is *discussed in a trivial way* by Callahan. In addition, it is not clear that the recursive construction leads to balanced work distribution.

```
kdtree P =
 if singleton P
 then Leaf P
 else
    let (cutdim, cutval, lpts, rpts) = split P
    in Node cutdim ctval (kdtree lpts) (kdtree rpts)
```

Figure 1. Recursive algorithm to build a k-d tree

4. Split by the median

Figure 1 shows a recursive algorithm for building a k-d tree on a set P of points. It is a classical recursion over the inductive definition of the k-d tree: when P has just one point, a leaf is returned; otherwise, a split is applied to produce a cutting discriminator (dimension) and value, a set of points that will live in the left subtree and a set of points that will go into the right subtree. Its realisation as a parallel algorithm demands:

- a parallel splitting process;
- unfolding of the recursive calls to avoid stages with idle processors;
- balancing the work load of the left and right trees construction.

4.1. Parallel splitting

Partitioning a set of points involves two decisions that can affect the height of the tree and load balance in a parallel environment: choice of the splitting dimension, and choice of the cut value or splitter.

Bentley's algorithm [8] cycles over the dimensions of the points to choose the splitting discriminator for each node. However, it is easy to show that cyclical choice of splitting dimension can contributing to an increased height of the tree: for example, if the points have a very narrow range for one or more dimensions. Moore [7], studying the use of k-d trees in processing neighbourhood search queries for robotic vision, shows interesting distributions where a cyclical choice of discriminators would have a negative effect on tree height search and build time. An example is the case where points are mostly distributed over the perimeter of an ellipse or circle. Experimental evidence collected by Omohundro [9] studying the behaviour of neural networks also points against the usage of cyclical choice of discriminators. It is not difficult to see, on the other hand, that splitting the dimension with widest range will in general contribute to balancing the tree. In this work the splitting dimension is always chosen as the one with widest range. The splitter in that dimension is chosen by either splitting close to the middle of the range or the median of the points' values in the chosen dimension.

Segdewick [12] shows a probabilistic study for Quicksort that favours the use of a median for splitting pivot choice. There are two options when choosing the splitter by the median: using Sedegwick's option of taking a median of three elements, or applying a deterministic efficient algorithm for median computation.

The algorithm published by Blum et al. in [13] has several interesting properties: it is a deterministic algorithm for selecting the kth element of a sequence; a parallel version can be easily obtained; it can be adapted to produce multiple splitters in one sweep. The algorithm is essentially composed of the following steps:

 (i) Group the sequence in blocks of 5 and find the median for each group.
 (ii) Apply recursion to find p, the median of the medians,

(iii) Use p to split the partition the sequence and use recursion on the appropriate subsequence to find the median (or the kth element).

The algorithm runs in $O(n)$ time when one processor is used. The first two steps almost invite parallelism to produce an algorithm that runs with $O(n)$ work with p processors. Those two steps produce a splitter that is good enough to keep the height of the tree within $O(\log n)$: the element yielded by them will always be greater than $3n/10$ and less than $7n/10$ elements of a given n elements sequence.

The last step in the split is moving the points themselves around the identified pivot. These is a problem that can be easily solved by a combination of a parallel map, that would identify where each points belongs, followed by a gather operation [14] or a sequence of two or more (in the case of a multi-way split) pack operations (compress in [15] and [16]).

4.2. Delayed node construction

The *split* function in algorithm 1 can now be completely parallel if decomposed in the following synchronous steps:

(i) in parallel find the dimension of split;

(ii) in parallel find the cut value;

(iii) in parallel move points to their destination partition.

The final hurdle in obtaining a parallel k-d tree construction is in the recursive step of the algorithm 1. A tail recursive algorithm could be easily converted to loop by hand or even by compilers of languages like LISP, or Haskell. As it stands, each call to build a *Node* has to be delayed to be fulfilled only after the two recursive calls to k-d tree are completed. The 2011 standard of C++ offers an elegant solution for that in the form of delayed computation through *futures*. Elegant as they may seem, they will still contribute to a general difficulty in work balance. An alternative approach is to make explicit both the stack of the recursive calls and the delayed computations, and produce a tail recursive descent through the tree.

The recursive descent through the tree will be traced by bookkeeping of:

- A list of subsets of points that are waiting to be partitioned. Each subset should be tagged with its parent node and the position where it belongs in that node.

- A list of delayed parent nodes: a parent node can be released from the delayed list when both its left and right children have been split and made into nodes.

In this frame, a parallel split applied to a *large* set of points produces:

- A delayed node, that will have to be updated with pointers to left and right children, when these are available.

- Two new sets of points that will go into the right and left children. These two sets of points will be tagged as belonging to left or right branches and will *know* their parent node.

Delayed nodes are promises of nodes that will be released (ready to be used) when pointers to their children become available.

Parallelism can be further improved by observing that splitting and recursive descent for smaller sets of points can benefit of special treatment, by processing them with a faster algorithm. It has been noticed [4] that Quicksort, for example, can be improved by resorting to Heapsort (when recursion becomes too deep) and insertion sort, when partitions become smaller.

The k-d tree algorithm has now the following phases:

- Synchronous phase: all processors repeatedly worked together to split one large set of points.

- Asynchronous phase: when a processor becomes available, it chooses any small set of points and builds a k-d tree over it.

points in 3-d	1 proc	4 procs	8 procs	12 procs
65536	$0.856s$	$0.868s$	$0.857s$	$0.854s$
252144	$3.206s$	$1.337s$	$1.235s$	$1.391s$
524288	$6.551s$	$2.545s$	$1.338s$	$1.512s$
786432	$9.724s$	$3.991s$	$2.781s$	$1.862s$
1572864	$18.43s$	$7.515s$	$4.532s$	$3.623s$
S_p	1	2.45	4.14	5.08
E_p	1	0.61	0.51	0.42

Table 1. Performance on a 12 cores machine

- Cross-references phase: release delayed nodes.

The cross-references phase can be dispensed with if nodes update, and possibly release, their parents as soon they are ready.

It's important to observe that the explicit representation of a delayed node can be dispensed with if the programming environment offers facilities to represent delayed computation: through channels and *goroutines* in the language Go, for example, or futures in C++.

5. Performance evaluation

Table 1 shows the results of performance tests for indexing sets of points in three dimensions. The testing environment is: machine with 24 GB of RAM, with dual Intel Xeon 5660 processor. Ubuntu 13.04 was the operating system used and the compiler was a gcc 4.7.3, wth OpenMP activated (gcc flag '-fopenmp').

Sets of points in the short queue were limited to 2^{16}, meaning that sets with less than 65537 points were processed by one processor. Times in the table are shown in seconds in columns by number of cores used. Each has times for sets of different sizes. Only one set with 2^{24} points was generated. The elements for each test were chosen randomly from that set.

The first line of the table shows essentially the same results for all columns because only one processor is being used, even when (last column) 12 processors are available.

The last two lines of the table show respectively speed-up and efficiency, based on their standard definitions:

- Speed-up $S_p = \frac{T_p}{T_1}$
- Efficiency $E_p = \frac{S_p}{p}$

6. Applications and related work

Both quad trees and k-d trees, if balanced, can be used to solve with $O(n \log n)$ work the problems introduced in section 1 of finding the closest pair of points and finding k nearest neighbours for all points in a set. The algorithms to solve those problems follow the recursive descent pattern described in section 1 and are detailed in [1]. Samet discusses previous work on parallel implementations of quad tree. Both Callahan and Har-Peled present algorithms that run in linear time to enumerate the set of all well separated pairs of points given a balanced k-d tree, in case of [10], or a balanced quad tree, in the case of [2]. Callahan [10] is possibly the most detailed theoretical study of the relation between k-d tree and the well separated pairs decomposition, and their realisation using parallel architectures. The technical literature doesn't seem show empirical results for implementations of either Har-Peled's or Callhan's algorithms.

The k nearest neighbours algorithm is especially attractive in HEP as it is found, for example, in TMVA [17], the toolkit for multivariate analysis, available in ROOT. Track reconstruction by

joining compatible triplets, used for example in CMS, can also benefit from the use of k-d trees, even when cellular automata computation is used to find the neighbourhoods, if dimensions higher than two are being considered. N-body computations based on Barnes-Hutt or Fast Multipole Method in general depend on tree methods [18]. N-point correlation functions have been tackled in Astronomy by the use of search in k-d trees [19].

7. Final remarks

Extensive testing shows that the algorithm performs remarkably well in achieving good balancing: for sets of 2^{24} points the height of the tree never exceeded 29 and that makes it already a good candidate for sequential implementation.

The parallel tests shows that efficiency should be improved. The present implementation uses garbage collected memory that degrades performance in the presence of any sort of concurrency. In addition, non-locality effects in memory, that can be seen in both copy and movement of keys that lie at the very heart of the algorithm, crucially affect speed-up when the number of processing elements is increased. Improvements can certainly be achieved by removal of garbage collected memory and, mainly, by careful allocation of memory for heap data structures closer to their respective cores.

8. Acknowledgment

References

[1] Samet H 2006 *Foundations of Multidimensional and Metric Data Structures* (Morgan Kaufman)
[2] Har-Peled S 2011 *Geometric Approximation Algorithms* (American Mathematical Society)
[3] Cormen T, Leiserson C, Rivest R and Stein C 2009 *Introduction to Algorithms* (MIT Press)
[4] Knuth D E 1998 *The Art of Computer Programming: Sorting and Searching* (Addison Wesley)
[5] Vaidya P M 1989 *Discrete and Computational Geometry*
[6] Finkel R A and Bentley J L 1974 *Acta Informatica* **4** 1–9
[7] Moore A W 1990 Efficient memory-based learning for robot control Tech. Rep. UCAM-CL-TR-209 University of Cambridge
[8] Bentley J L 1975 *Communications of ACM* **18** 509–517
[9] Omohundro S M 1987 *Journal of Complex Systems* **1** 273–347
[10] Callahan P B 1995 *Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and Its Applications* Ph.D. thesis John Hopkins University
[11] Bentley J L 1999 *Programming Pearls* (Addison Wesley)
[12] Sedgewick R 1978 *Communications of ACM* **4** 847–857
[13] Blum M, Floyd R W, Pratt V R, Rivest R L and Tarjan R E 1973 *Journal of Computer and System Sciences* **7** 448–461
[14] McCool M, Robinson A D and Reindeers J 2012 *Structured Parallel Programming: Patterns for Efficient Computation* (Morgan Kaufman)
[15] Blelloch G E 1990 Prefix sums and their applications Tech. Rep. CMU-CS-90-190 School of Computer Science – Carnegie Mellon University
[16] Akl S G 1989 *The design and analysis of parallel algorithms* (Prentice-Hall)
[17] Hoecker A *et al.* 2007 *PoS* **ACAT** 040 arXiv:physics/0703039
[18] Pfalzner S and Gibbon P 1996 *Many-body Tree Methods in Physics* (Cambridge University Press)
[19] Moore A, Connolly A, Genovese C, Gray A, Grone L *et al.* 2000 (*Preprint* `astro-ph/0012333`)