CSE 392 Spring 2014 Homework 3

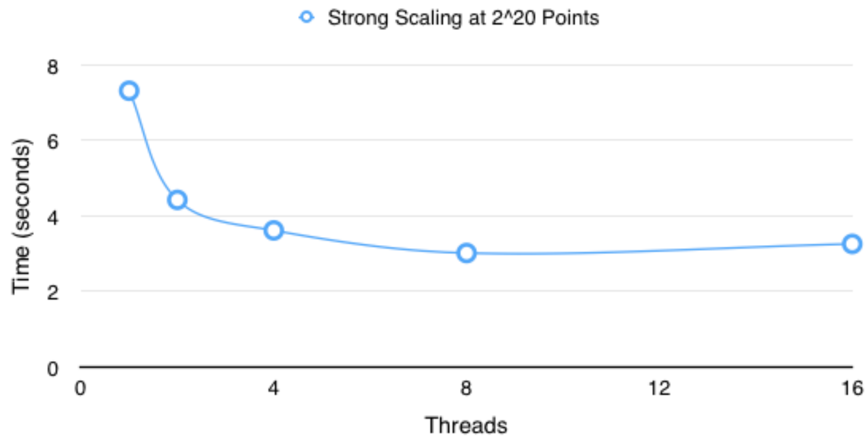
Abhishek Bhaduri and Aaron Stacy

1. See appended hand-written work
2. 2D N-body Simulation
    1. The main function of the algorithm, `nbody`, follows the pseudo-code given in lecture 15 slide 31 fairly closely:

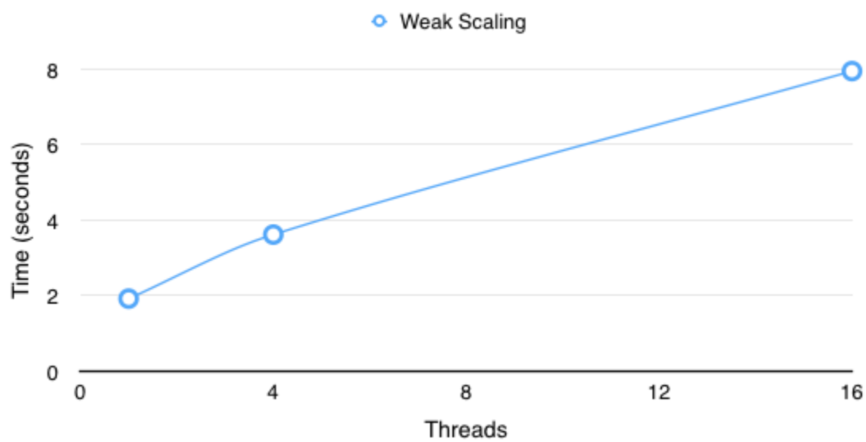
```
function nbody(pointsAndDensities, n, outputPotential)
  parfor i=1-n                              % W=N, D=1
    mids[i] = convertToMid(points[i])
  end

  [smids,idx] = parallelSort(mids)          % W=NlogN, D=logN * loglogN

  trees = []
  lengths = []

  parfor i=1:p                              % W=NlogN, D=logN
    myStart = n / p * i
    myEnd = myStart + n / p
    trees{i} = qtree()
    for j=myStart:myEnd
      trees{i}.insert(points[j])
    end
    lengths[i] = length(trees{i})
  end

  tree = []

  parfor i=1:p                              % W=N, D=1
    tree[sum(lengths(0:i-1))] = trees{i}.preOrder
  end

  tree = parallelSort(tree)                 % W=NlogN, D=logN * loglogN

  tree = removeDuplicates(tree)             % W=N, D=1

  [i, o] = eulerTour(tree)                  % W=N, D=logN

  treePrefixScan(tree, i, o, density)       % W=N, D=logN

  parfor i-1:N                              % W=NlogN, D=logN
    outputPotential[i] = evaluate(points(i), tree.root)
  end
end
```

For parallel sort we used Intel TBB. For converting to morton ID's, inserting into the tree, and evaluating points we used the algorithms from the given Matlab implementation (`body.cpp`, `qtree.cpp`, and `euler.cpp`). For doing the Euler tour and prefix scan on the tree we used the algorithm from slide 28 of lecture 15.

2. Scalability results:
    1. Strong scaling at ~ 1 million points



2. Weak scaling at a ratio of $2^{18} : 1$



3. We could efficiently estimate error by experimentally measuring the average error that center-of-mass approximation introduces, and then using that as a heuristic during the last step of the algorithm. This adds an $O(N)$ time step to the end of the algorithm.

Morton Orderings

Notes:

(1) Assume K-D space (ex: K=2 for quad trees, K=3 for oc trees etc.)

(2) Any point will need K coordinates (ex: $x, y, z, \ldots$)

(3) We are using binary representation of the Morton IDs

(4) For each level in the tree, we will need 1 bit per coordinate

(5) For L levels, we need L bits per coordinate (ex: L bits for x, L bits for y ... etc.)

(6) For all 'K' coordinates we need $L * K$ bits to represent a point

(7) For L levels we need $\lceil \log L \rceil$ bits

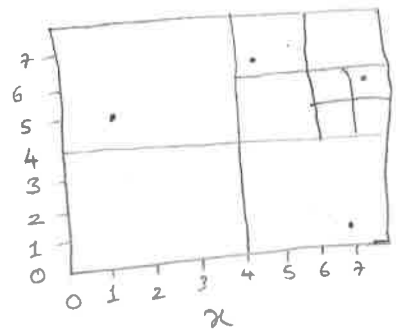
(8) Total size of Morton ID will be $[(L * K) + \lceil \log L \rceil]$ bits

(9) For any node, given its anchor (say, lower left corner) and its level, we can form the Morton ID by bit interleaving and appending the level.

(10) When sorted, Morton IDs of parents appear before IDs of the children (pre-order traversal)

Morton ID $\underbrace{\boxed{x, y, z, \ldots \qquad\qquad}}_{\text{Base}}\underbrace{\boxed{\qquad\quad}}_{\text{Level}}$

Sample Quad Tree



A point is (x, y)

Some examples from this picture —

(a) root @ (0,0) level ∅

(b) node @ (4,4) level 1

(c) node @ (4,4) level 2

(d) node @ (7,5) level 3

1 Determine ancestor

Given nodes n_1 & n_2 at levels l_1 & l_2 respectively.

We know $l_1 < l_2$

Morton ID of n_i is $\boxed{b_i \mid l_i}$ $= MID_i$

↙ base ↘ level

We have —

For n_1 to be an ancestor of n_2

First $(l_1 * K)$ bits of MID_1
$= =$ First $(l_1 * K)$ bits of MID_2

and,

Remaining bits $((L*K) - (l_1*K))$ of $MID_1 ==$ all zeros

This is an $O(1)$ computation.

Ex: from the picture, say n_1 @ (6,4) level 2
n_2 @ (7,5) level 3

$\begin{pmatrix} 110, 100 \text{ @ } 10 \\ 111, 101 \text{ @ } 11 \end{pmatrix}$

base b_i — level l_i

$MID_1 = 110100 \quad 10$

$MID_2 = 110111 \quad 11$

$K = 2$ for quad tree

$l_1 = 2$ ⟶ So, first $l_1 \times K = 4$ bits of two morton IDs are 1101

$l_2 = 3$ And, remaing bits of MID_1 base are all zeros.

We can extend this to octrees or any other value of K

2 Find colleagues of node n_1

Say n_1 is at level l_1

In the octree a node has 8 children (2^K with $K=3$)

For any node n_1 all siblings at l_1 are colleagues.

These colleagues can be enumerated by their morton IDs with values 0 to 2^K-1

If we sweep the bits $((l_1-1)*K)+1$ ~~to~~ to $(l_1 * K)$

we get morton IDs of all siblings.

We can then do parallel binary search on all IDs to get the corresponding nodes.

This will give $\underline{O(\log n)}$ if we search on $P = 2^K - 1$ for octrees

#3 Least Common Ancestor

Given nodes n_1 & n_2
with levels $l_1 \geq l_2$

We assume, parents are at a lower level than children.

Algorithm

(1) Find ancestor of n_1 @ level l_2 and assign to n_1
if $l_1 > l_2$ else ignore.

(2) Now that both n_1 & n_2 are at level l_2 we can use the bit manipulation technique used in #1 for finding ancestors —

(a) we can XOR morton IDs of n_1 & n_2

(b) the string of ϕ's at the MSB end represent MSB's that are equal between MID_1 & MID_2

(c) These MSBs represent the bits that belong to the ancestor with remaining bits equal to all zeros.

This is an $O(1)$ computation.

#4 Rank of Children

* Rank of element a_i is
  # of elements in A $\leq a_i$

* Morton IDs of children is > Morton ID of parent
  (LSBs get added and level increases)

Given (a) array $A(N)$ with elements a_i = morton ID of node n_i
$\forall\ i = 0, \cdots, N-1$

(b) array is sorted.

In the sorted array, the node after the parent can be —

1. child
2. sibling
3. node from different ancestor.

We will check for ancestor-child relationship between node i and all entries following i (same technique as #1)

```
rank_list = []
ref = a(i) , ref_rank = rank(a(i))
i++
while (is-ancestor (ref, a(i)) ) {
    rank_list.append (++ ref-rank)
    i++
}
// loop breaks when we run into a non-child.
// loop termination needs to accomodate size-of-array (not shown).
```

The loop is sequentially $O(N)$ but it can be embarrassingly parallel to approach depth $O(1)$