Accelerators to Applications
Algorithms Lesson

Nearest-Neighbors

Kyle Burke
Assistant Professor of Computer Science
Wittenberg University
Summer 2011

Summary

This lesson covers two parallel solutions to the Nearest-Neighbors problem (also known as the Closest-Pair problem) on the two-dimensional plane. The goal of this algorithm is to return the pair of points from some input set that have the shortest distance between them. The solutions include a brute-force algorithm and a divide-and-conquer algorithm. The serial running times of these algorithms are $\Theta(n^2)$ and $\Theta(n \log(n))$, respectively, but both run in $\Theta(n)$ time in parallel with n processors. This highlights the effectiveness of parallel programming as compared to serial algorithm design.

Prerequisites

Students need the following background before starting this lesson:

- The "Theory of Parallel Algorithms" A2A lesson.
- A description of the serial solutions to Nearest-Neighbors.
- An understanding of at least one parallel sorting algorithm, as described in the "Speedy Sorting" A2A lesson.

Outcomes

After this lesson, students will have gained:

- Knowledge of the parallel steps and complexity analysis of both algorithms.
- Experience applying parallel algorithm analysis to geometric algorithms.
- Understanding of the benefits of parallel programming with both brute-force and divide-and-conquer algorithms.
- Knowledge of the trade-offs between more involved serial algorithm design and parallel programming concepts.

Required Resources

This lesson can be taught with only a markerboard or chalkboard and the proper writing utensils.

Lesson Outline

This lesson includes:

- A description of how to parallelize each of the two Nearest-Neighbors algorithms.
- Analysis of the parallel running times, using Brent's Theorem and n processors.
- Both of the algorithms use `distanceBetween` (see Algorithm 1) but it is probably not necessary to write out for students.

Algorithm 1 `distanceBetween`

Finds the distance between two points in two-dimensions.
Input: $p0$ and $p1$, real arrays of length 2.
Output: $distance$, the distance between $p0$ and $p1$.

1: **return** $\sqrt{(p0[0] - p1[0])^2 + (p0[1] - p1[1])^2}$

Brute-Force Algorithm

- Recall: the serial brute-force algorithm computes the distance between all pairs of points to find the nearest neighbors. Thus, it requires $\Theta(n^2)$ time to find the nearest neighbors from n points.
- This algorithm can be parallelized by replacing `for` with `for all` in a few loops. The code is laid out in Algorithm 2.
- Analysis using Brent's Theorem: $TP(n, n) \in \Theta(n)$
- This is a speedup factor of n from the serial algorithm.

Divide-and-Conquer Algorithm

- Recall: the analysis of the serial divide-and-conquer algorithm is somewhat more involved.
- Design of the main algorithm:
  - To simplify the content of the main algorithm, we provide two additional algorithms to be called inside `nearestNeighbors`. One of these, `sortPoints` sorts the points (we recommend that the Speedy Sorting lesson be covered apriori) based on either the x or y coordinate. The interface for this algorithm is given in Algorithm 3. The steps of the algorithm are omitted for simplicity.
  - Additionally, the algorithm `getPointsNearVerticalLine` is used. The details of this algorithm are given in Algorithm 4. Students may not need to see the details of this algorithm either.
  - Since `getPointsNearVerticalLine` is a serial algorithm, it will return a list of points that is still sorted, assuming p was sorted to begin with. For this reason, the `for` loop in `getPointsNearVerticalLine` should not be parallelized.

Algorithm 2 `nearestNeighbors` (Parallel brute-force version)

Finds the closest pair of points in a two-dimensional space.
Input: An array of n points, $p[0..n-1]$.
Output: A 2-tuple of points, $closestPair$.

```
1: dists[0..n-1][0..n-1]
2: for all i ∈ [0..n-1] do
3:     for all j ∈ [0..n-1] do
4:         if i < j then
5:             dists[i][j] ← distanceBetween(p[i], p[j])
6:         else
7:             dists[i][j] ← ∞
8:         end if
9:     end for
10: end for
11: minDists[0..n-1]
12: closestPoint[0..n-1]
13: for all i ∈ 0..n-1 do
14:     minDist[i] ← ∞
15:     closestPoint[i] ← ∞
16:     for j ∈ 0..n-1 do
17:         if minDists[i, j] < minDist[i] then
18:             minDist[i] ← minDists[i, j]
19:             closestPoint[i] ← j
20:         end if
21:     end for
22: end for
23: closestPair ← (p[0], p[1])
24: minDistance ← distanceBetween(p[0], p[1])
25: for i ∈ 0..n-1 do
26:     if minDist[i] < minDistance then
27:         minDistance ← minDist[i]
28:         closestPair ← (p[i], p[closestPoint[i]])
29:     end if
30: end for
31: return closestPair
```

Algorithm 3 `sortPoints`

Returns a sorted copy of a list of points.
Input: An array of 2-dimensional points, p and the index for which gets sorting priority (0 for x, 1 for y).
Output: $sorted$, a sorted copy of p.

Algorithm 4 `getPointsNearVerticalLine`

Filters out points not near a vertical line in two-dimensional space.
Input: An array of length-2 arrays, p, the x-value of a vertical line, x, and the distance from the line of points desired, $distance$.
Output: $nearPoints$, an array of length-2 arrays representing those points in p within $distance$ from the vertical line with y-intercept, x.

```
1: nearPoints[]
2: for i ∈ {0, ..., length(p)} do
3:     if |p[i][0] − x| ≤ distance then
4:         append p to nearPoints
5:     end if
6: end for
7: return nearPoints
```

  - Next, the `nearestNeighbors` algorithm itself should be described. There is only a small change: to run the two recursive calls simultaneously. Nevertheless, the algorithm is fully given here as Algorithm 5.

- The analysis is not especially difficult, despite the length of the code.
  - There are a constant number of statements up until the parallel recursive calls at line 21.
  - After the recursive calls, $\Theta(n)$ steps will be executed (not $\Theta(n^2)$ since the inner `for` loop is run a maximum of 7 times) including the calls to `getPointsNearVerticalLine` and `sortPoints`.
  - The recurrence relation and solution for the work is:

$$\begin{aligned} W(n) &= 2W(n/2) + \Theta(n) \\ &= \Theta(n \log(n)) \end{aligned}$$

  - And for the step complexity:

$$\begin{aligned} S(n) &= S(n/2) + \Theta(n) \\ &= \Theta(n) \end{aligned}$$

  - Using Brent's Theorem for the time complexity:
    - For any number of processors, p:

$$TP(n,p) \in \Omega\left(\left\lceil \frac{n\log(n)}{p} \right\rceil\right)$$

and

$$\begin{aligned} TP(n,p) &\in O\left(\frac{n\log(n)}{p} + n)\right) \\ &= O\left(n\left(\frac{\log(n)}{p} + 1\right)\right) \end{aligned}$$

Algorithm 5 `nearestNeighbors` (Parallel divide-and-conquer version)

Finds the closest pair of points in a two-dimensional space.

Input: An array of points, $p[low..high]$, sorted by x coordinates in increasing order, as well as integers low and $high$, indices in the array P.

Output: A 2-tuple of points, $closestPair$.

```
1: n ← 1 + high − low
2: if n = 2 then
3:     closestPair[0] ← p[low]
4:     closestPair[1] ← p[high]
5: else if n = 3 then
6:     distance01 ← distanceBetween(p[low], p[low + 1])
7:     distance02 ← distanceBetween(p[low], p[high])
8:     distance12 ← distanceBetween(p[low + 1], p[high])
9:     if distance01 < distance02 and distance01 < distance12 then
10:        closestPair[0] ← p[0]
11:        closestPair[1] ← p[1]
12:    else if distance02 < distance01 and distance02 < distance12 then
13:        closestPair[0] ← p[0]
14:        closestPair[1] ← p[2]
15:    else
16:        closestPair[0] ← p[1]
17:        closestPair[1] ← p[2]
18:    end if
19: else
20:    middle ← (low + high)/2
21:    SimultaneouslyDo
22:        leftClosest ← nearestNeighbors(p[low, middle], low, middle)
23:        rightClosest ← nearestNeighbors(p[middle + 1, high], middle + 1, high)
24:    EndSimultaneous
25:    minDistance ← distanceBetween(leftClosest[0], leftClosest[1])
26:    closestPair ← leftClosest
27:    if distanceBetween(rightClosest[0], rightClosest[1]) < minDistance then
28:        minDistance ← distanceBetween(rightClosest[0], rightClosest[1])
29:        closestPair ← rightClosest
30:    end if
31:    sortedByY ← sortPoints(p, 1)
32:    pointsAlongLine ← getPointsNearVerticalLine(sortedByY, p[middle][0], minDistance)
33:    for i ∈ {low, ..., high − 1} do
34:        for j ∈ {i + 1, ..., min(i + 7, high) do
35:            distance ← distanceBetween(pointsAlongLine[i], pointsAlongLine[j])
36:            if distance < minDistance then
37:                minDistance ← distance
38:                closestPair[0] ← pointsAlongLine[i]
39:                closestPair[1] ← pointsAlongLine[j]
40:            end if
41:        end for
42:    end for
43: end if
44: return closestPair
```

* And when $p = n$:

$$TP(n, n) \in \Omega(\log(n))$$
and
$$TP(n, n) \in O(n)$$

  - The speedup attained in the case with n processors is a factor of $\log(n)$.

- **Key Point**: Even in a complicated divide-and-conquer algorithm, the basic tactic of running the recursive calls in parallel can work. In this case, there is extra work to be done to use parallel sorting as well. (Indeed, before calling the algorithm, a parallel sort must be used to sort the points by their x-coordinate.
- **Key Point**: The parallel divide-and-conquer algorithm is much more complicated to implement. If you know you will have on the order of n processors available, it may be worthwhile to implement the brute-force algorithm to save time on the programming end. The brute-force algorithm will behave much more poorly in the presence of few processors, however.