Sorting Algorithms

Joseph Randall Hunt
Bobby Wertman
Western Carolina University,
Cullowhee, North Carolina

December 3, 2011

1 Project Proposal

1.1 Goals

1.2 Plan

2 Sorts

2.1 Merge Sort

2.1.1 Algorithm

Merge sort is a comparison-based sorting algorithm, based on the divide-and-conquer design. Its average and worst cases are both $nlog(n)$, and its best case is $\Omega(n)$. Invented in 1945 by John von Neumann, it exploits the fact that combining two lists of sorted data is a linear-time process. [1]

2.1.2 Efficiency

The actual code for the algorithm is available in **Listing 1**. This particular implementation uses an optimization that switches to insertion sort on small arrays. This speeds up the algorithm because it allows the small data set to fit entirely in cache along with the small amount of code associated with the insertion sort algorithm. See **Listing 2** for a parallel implementation of merge sort.

2.1.3 Applications

Merge sort is useful in applications where the data set will not fit entirely into memory. This allows for the data to be read in from disk and sorted as it is read, thus requiring a very small memory footprint. In addition, when time complexity needs to be guaranteed, merge sort is preferable over quicksort, as quicksort's worst case is $O(n^2)$.

2.2 Quick Sort

2.2.1 Algorithm

2.2.2 Efficiency

2.2.3 Applications

2.3 Shell Sort

2.3.1 Algorithm

2.3.2 Efficiency

2.3.3 Applications

2.4 Comparing and Contrasting

3 Conclusions

A Code Examples

Listing 1: Go implementation of MergeSort

```go
package main

var optimize int = 10

func Merge(arr []int) {
  if len(arr) < optimize && len(arr) != 1 {
    InsertionSort(arr)
  } else if len(arr) > 1 {
    // Start at the midpoint
    size := len(arr) / 2

    left  := arr[:size]
    right := arr[size:]

    // Perform sub-merges
    Merge(left)
    Merge(right)

    // For holding results
    data := make([]int, len(arr))

    i := 0
    for len(left) > 0 && len(right) > 0 {
      // Decide which list to pull from
      if left[0] > right[0] {
        // Put the first value into the results and shrink the array
        data[i], right = right[0], right[1:]
      } else {
        // Same thing for the left
        data[i], left = left[0], left[1:]
      }
```

```go
      i++ // Next element
    }

    // Figure out which list isn't empty
    var remainder []int

    if len(left) > 0 {
      remainder = left
    } else {
      remainder = right
    }

    // Empty remaining values
    for len(remainder) > 0 {
      data[i], remainder = remainder[0], remainder[1:]
      i++
    }

    // Copy result to our answer
    for i := 0; i < len(arr); i++ {
      arr[i] = data[i]
    }
  }
}

func InsertionSort(arr []int) {
  for i, j := 1, 1; i < len(arr); i, j = i+1, i+1 {
    for j > 0 && arr[j] < arr[j-1] {
      arr[j-1], arr[j] = arr[j], arr[j-1]
      j--
    }
  }
}
```

Listing 2: Parallel implementation of MergeSort in Go

```go
package main

import "sync"

var optimize int = 25

func Merge(arr []int, group *sync.WaitGroup) {
  if len(arr) < optimize && len(arr) != 1 {
    InsertionSort(arr)
  } else if len(arr) > 1 {
    // Start at the midpoint
    size := len(arr) / 2

    left  := arr[:size]
    right := arr[size:]

    // Perform sub-merges
    group := new(sync.WaitGroup)
    group.Add(2)
      go Merge(left,  group)
      go Merge(right, group)
```

```go
    group.Wait()

    // For holding results
    data := make([]int, len(arr))

    i := 0
    for len(left) > 0 && len(right) > 0 {
      // Decide which list to pull from
      if left[0] > right[0] {
        // Put the first value into the results and shrink the array
        data[i], right = right[0], right[1:]
      } else {
        // Same thing for the left
        data[i], left = left[0], left[1:]
      }
      i++ // Next element
    }

    // Figure out which list isn't empty
    var remainder []int

    if len(left) > 0 {
      remainder = left
    } else {
      remainder = right
    }

    // Empty remaining values
    for len(remainder) > 0 {
      data[i], remainder = remainder[0], remainder[1:]
      i++
    }

    // Copy result to our answer
    for i := 0; i < len(arr); i++ {
      arr[i] = data[i]
    }
  }
  group.Done()
}

func InsertionSort(arr []int) {
  for i, j := 1, 1; i < len(arr); i, j = i+1, i+1 {
    for j > 0 && arr[j] < arr[j-1] {
      arr[j-1], arr[j] = arr[j], arr[j-1]
      j--
    }
  }
}
```

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, Mass, 2nd edition,

2001.