An Analysis of Multiprocessor Quicksort

Presented By:
Kevin Ellis, Braden Harrelson, Ryan Luig

What is a Serial and Parallel Program?

- A serial program utilizes one processor to accomplish its tasks
- A parallel program utilizes multiple processors in order to complete multiple tasks simultaneously

Project Goals

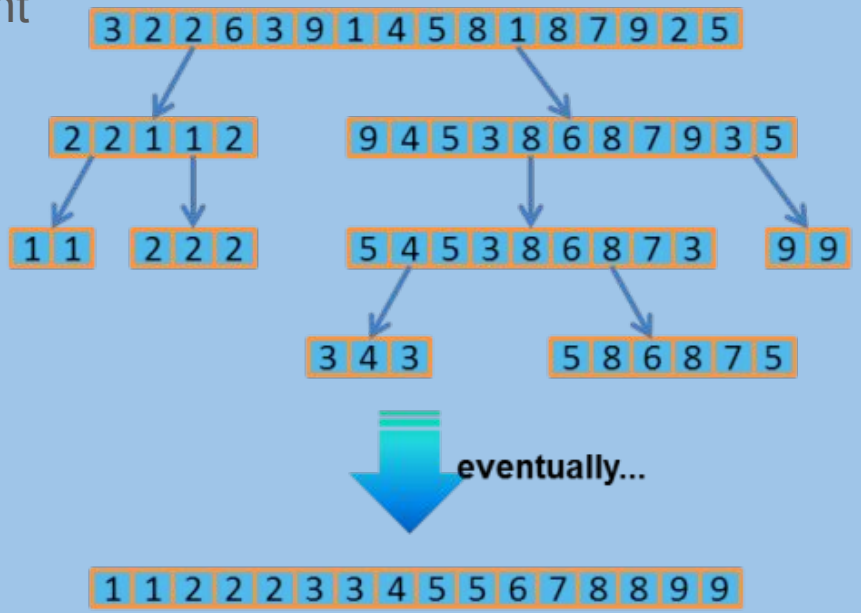
- Businesses store large amounts of data
    - client records, transaction list, etc.
- Develop sorting algorithm using MPI (Parallel library)
- Increase speed while being weary of memory usage
- Compare speed of serial vs parallel

Quicksort Algorithm vs Other Algorithms

- Quicksort is in place
  - less memory usage
- Complexity
  - Average complexity of O(nlogn)
  - worst case $O(n^2)$

How Serial Quicksort Works

- Select a pivot (We select the leftmost element)
- If less move to the left, more to the right
- Recursively call on left and right sides



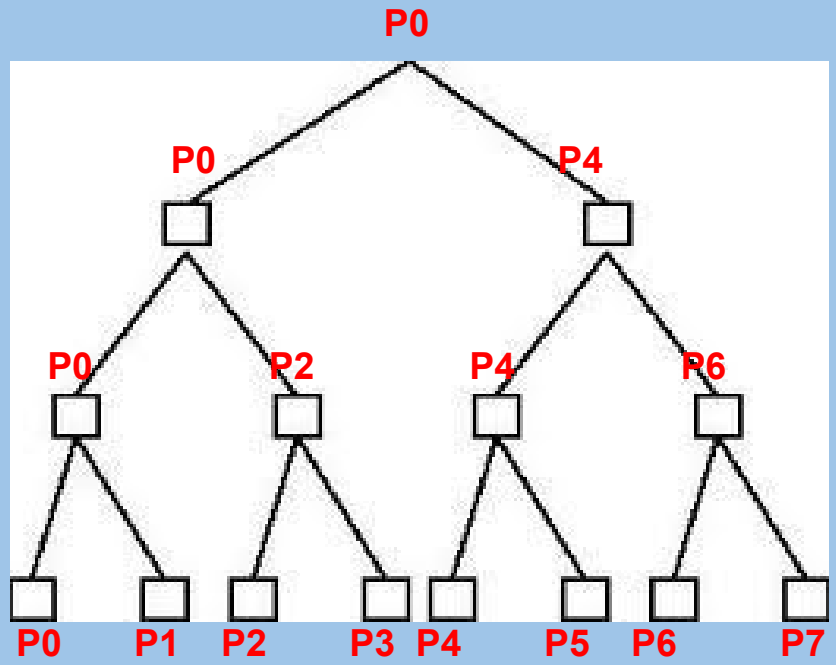
(Pic source: hpcwire.com)

Initial Approaches

- Sample sort
  - Issues with sentinel values
- Recursive pivot moving
  - Communication issues

Parallel Approach

- Divide and Conquer
- Sort Partition and Send
- Receive and Merge
- Repeat



Parallel Approach Breakdown

- Set a step value equal to 1
- Loop from step value to number of processors
  - if(Process Id % (step * 2)):
    - Receive a recvsize from its partner
    - Allocate memory of recvsize
    - Receive a sorted array from the it’s partner; Store it in the just allocated memory
    - Call mergeSubset with two arrays [process local array] and [the array just recvd]
    - Allocate memory of recvsize * 2 to store the array returned by mergeSubset
  - else:
    - Send your size to it’s partner
    - Send your locally sorted array to it’s partner
  - Increment step by step*2
- Break out of the loop

Merge Subsets

- Input:Two Sorted Subsets, Size of Subset
- Output: One Sorted Set
- Compare Elements in Subsets

Set1 { 1, 7, 11, 12, 15, 34, 35, 36}

Set2 { 2, 3, 4, 10, 13, 20, 21, 22}

Output {1, 2, 3, 4, 7, 10, 11, 12, 13,15, 20, 21, 22, 34, 35, 36 }

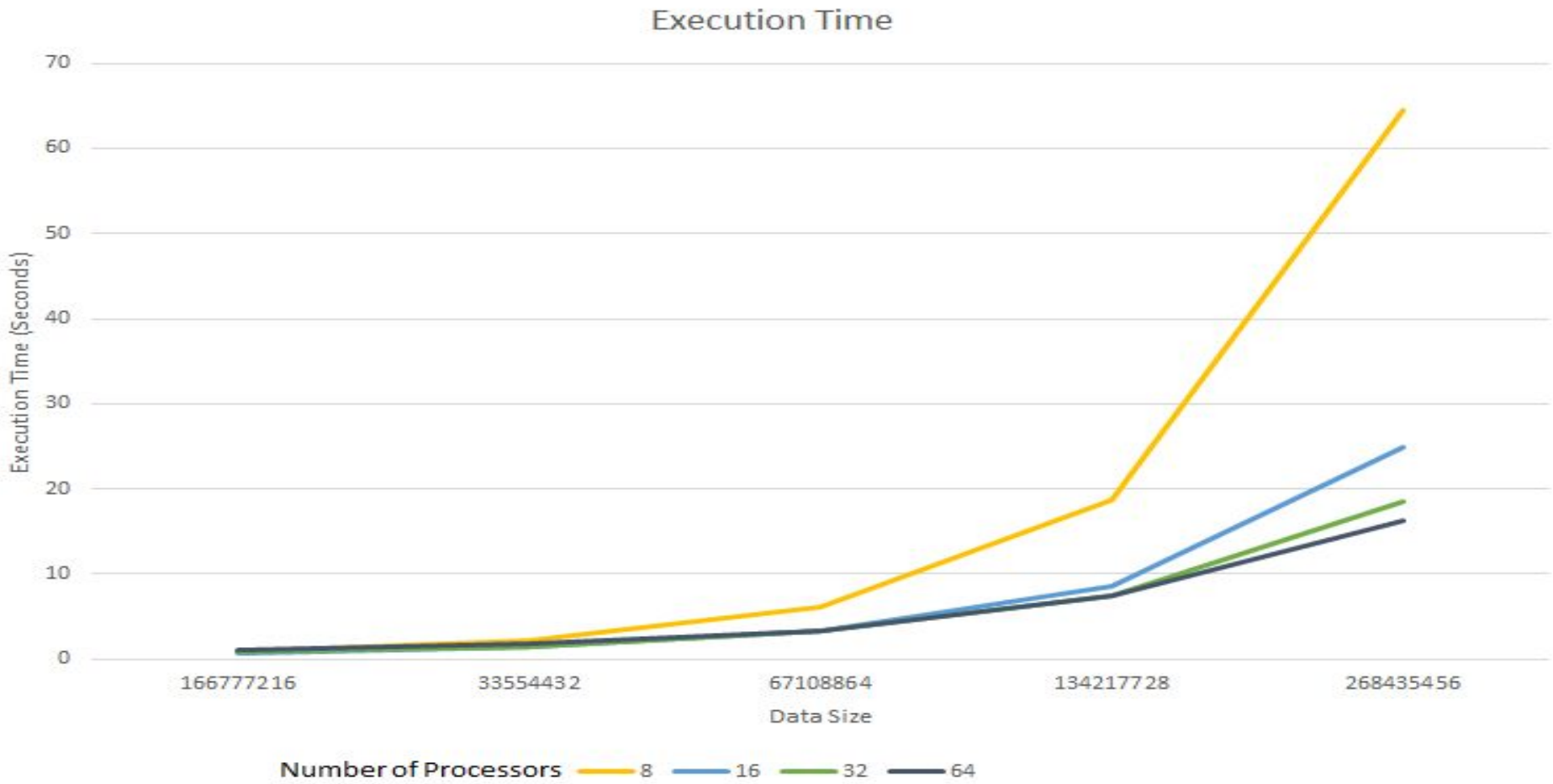
Data Breakdown

- Both Serial Parallel programs read from a file of random numbers.

  IMPORTANT NOTE: Since they read from a file they read the EXACT same set of numbers this is extremely important in determining the relative speed among the Serial and Parallel algorithms

- Each of these files contain 2^x random numbers
  - In this presentation we are using data sets of size
    - 16,777,216 (2^{24})
    - 33,554,432 (2^{25})
    - 67,108,854 (2^{26})
    - 134,217,728 (2^{27})
    - 268,435,456 (2^{28})

Note: To sequentially sort the data size 2^{28} it takes 2989 seconds or 49 minutes and 48 seconds



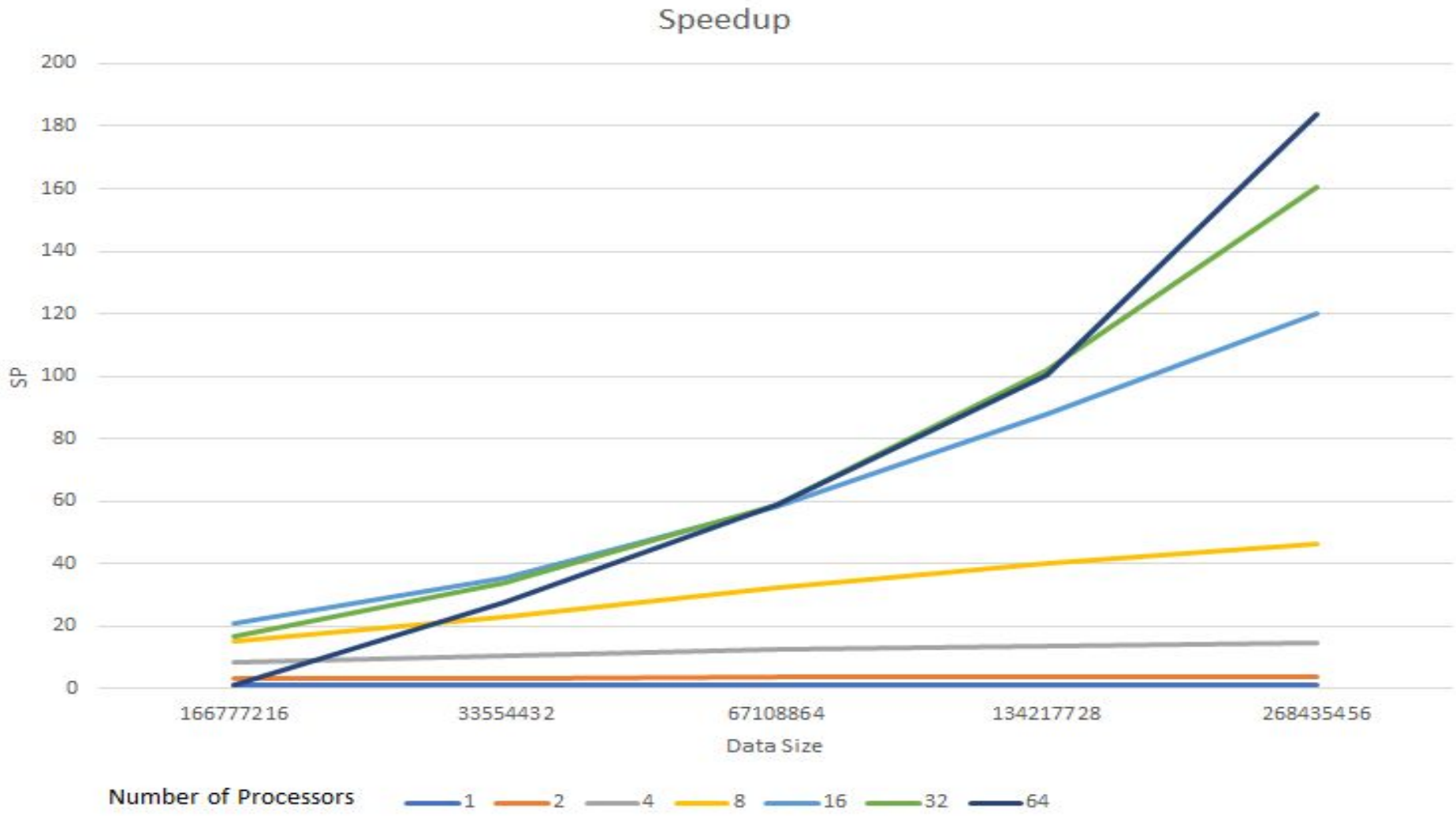
Note: Where it previously took 49 minutes and 48 seconds for 2^{28} numbers to be sorted sequentially
Utilizing multiple processors can sort in a fraction of a time - FOR EXAMPLE: it takes 16 processors 24.88 seconds

What is Speedup?

Speedup provides an answer to the question of how much faster can our multiprocessor implementation sort the given data set

Speedup = Ts / Tp

- Ts : Execution time of the serial approach
- Tp : Execution time of the parallel approach

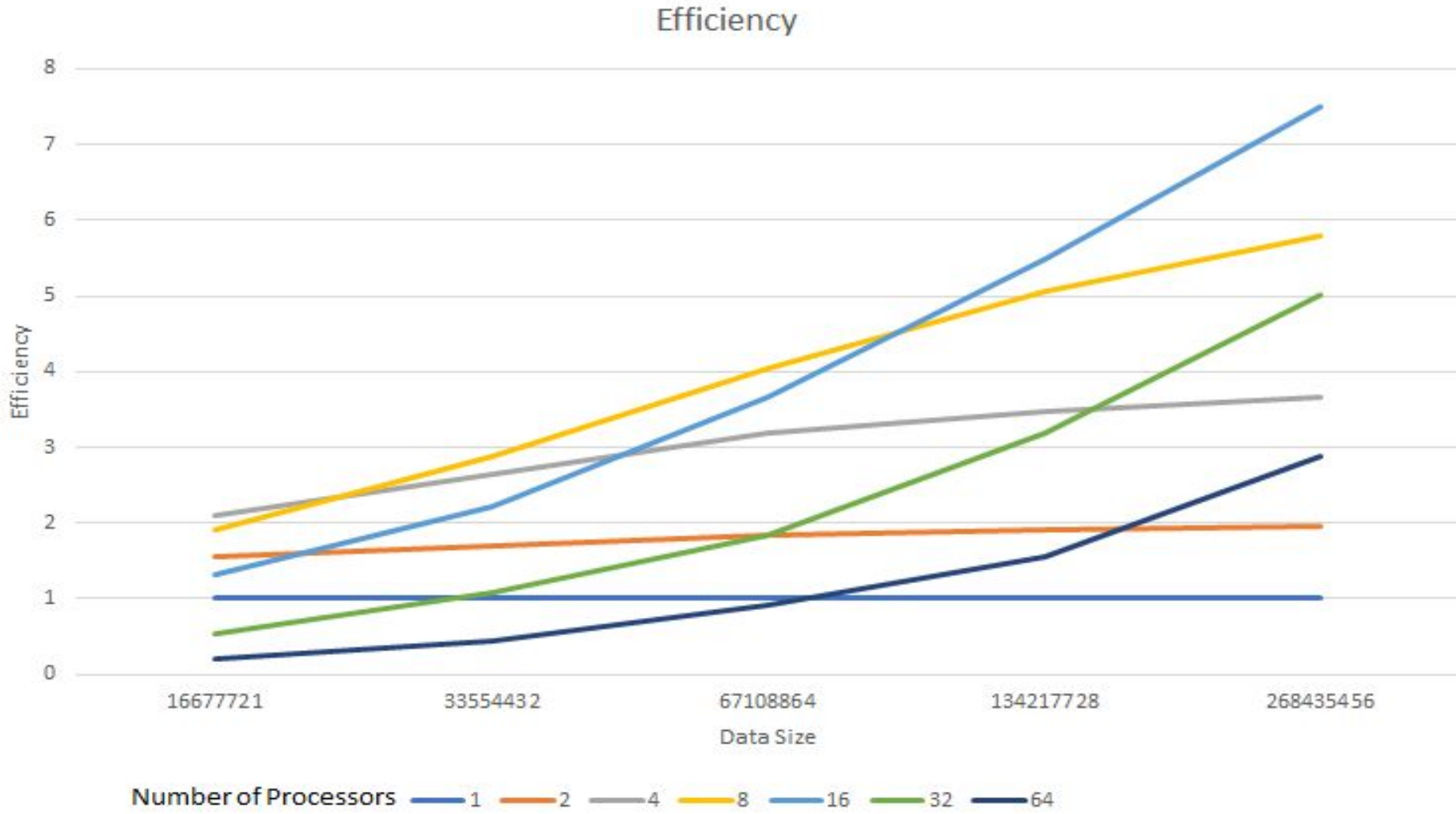

Speedup
200
180
160
140
120
100
80
60
40
20
0
SP
166777216
33554432
67108864
134217728
268435456
Data Size
Number of Processors
1
2
4
8
16
32
64

What is Efficiency?

Efficiency tells us how long each individual processor is being used on computation

Efficiency = Ts / (Number of Processors * Tp)

- Ts : Execution time of the serial approach
- Tp : Execution time of the parallel approach


Efficiency
8
7
6
5
4
3
2
1
0
Efficiency
16677721
33554432
67108864
134217728
268435456
Data Size
Number of Processors
1
2
4
8
16
32
64

Review

- Why do we need a parallel sort?
  - Large data sets
- Divide and conquer datasets
- Results
  - Execution times
  - Speed up
  - Efficiency

Thank You! Any Questions?