CSCI 251: Concepts of Parallel and Distributed Systems

Alvin Lin

September 2017 - December 2017

Project 2

This project involved the implementation of a serial and parallel bitonic sort. This implementation was done serially in C and parallelized using POSIX threads. The parallel implementation of the algorithm simply divided the recursive operation among the specified number of threads.

Serial Psuedocode

This implementation assumed that the given data elements to sort were integers and that the size of the data set was a power of two. In addition, the threaded implementation assumes that the number of threads to distribute this among is a power of two.

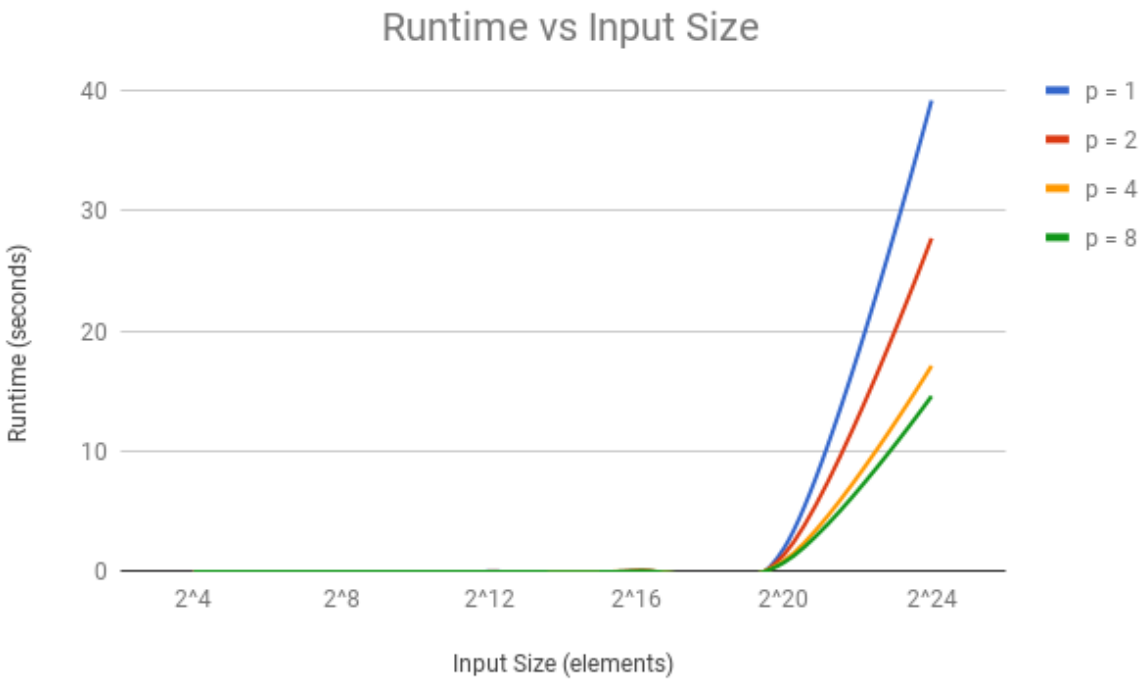
```
def compare_swap(data, direction, i, j):
  if direction < 0 and data[i] < data[j] or
    direction > 0 and data[i] > data[j]:
    swap(data[i], data[j])

def bitonic_merge(data, start, end, direction):
  if end - start <= 1:
    return
  int k = (start + end) / 2
  for i = start, j = k; i < k; ++i, ++j:
    compare_swap(data, direction, i, j)
  bitonic_merge(data, start, k, direction)
  bitonic_merge(data, k, end, direction)

def bitonic_sort(data, start, end, direction):
  if end - start <= 1:
    return
  let k = (start + end) / 2
  bitonic_sort(data, start, k, 1)
  bitonic_sort(data, k, end, -1)
  bitonic_merge(data, start, end, direction)
```

Analysis

The full data sheet of the runtimes is in the included data.pdf file. In general, the runtime improved the more threads we used.



At low data sizes though, the extra threads caused overhead and led to a decrease in speedup. In general however, more threads allowed for better performance at high input sizes. The marginal benefit gained by adding more threads decreased the more threads were added.