Pune Institute of Computer Technology Dhankawadi, Pune

A MINI PROJECT REPORT ON

Implement Merge Sort and Multithreaded Merge Sort

SUBMITTED TO THE SAVITRIBAI PHULE PUNE UNIVERSITY, PUNE IN THE PARTIAL FULFILLMENT OF THE LP III

BE (COMPUTER ENGINEERING) SUBMITTED BY

Roll No.	Name
41471	Said Prachi Sachin

Under the Guidance of

Prof. S. P. Shintre

DEPARTMENT OF COMPUTER ENGINEERING

Academic Year 2024-25

PUNE INSTITUTE OF COMPUTER TECHNOLOGY DEPARTMENT OF COMPUTER ENGINEERING

CERTIFICATE

This is to certify that the Mini Project report of LP III (DAA) entitled

”Implement merge sort and multithreaded merge sort” Submitted by

Roll No.	**Name**
41471	Prachi Said

has satisfactorily completed a micro project report under the guidance of Prof. S. P. Shintre towards the partial fulfillment of BE Computer Engineering, Academic Year 2024-25 of Savitribai Phule Pune University.

Prof. S. P. Shintre
(Lab Guide)

Dr. Geetanjali Kale
(H. O. D)

Date: / / Place: Pune

Introduction

Implement merge sort and multithreaded merge sort. Compare the time required by both algorithms. Also, analyze the performance of each algorithm for the best case and the worst case.

Objective

- Implement merge sort using multi-threading.
- To analyze the performance of the multi-threading approach on merge sort.
- To analyze the complexity performance of the multi-threading approach on merge sort

Scope

Background jobs like running application servers like Oracle application servers, and Web servers like Tomcat, etc which will come into action whenever a request comes.

1. Performing some execution while I/O blocked.
2. Gathering information from different web services running in parallel.
3. Typing MS Word documents while listening to music.
4. Games are very good examples of threading. You can use multiple objects in games like cars, motorbikes, animals, people, etc. All these objects are nothing but just threads that run your game application.
5. Railway ticket reservation system where multiple customers access the server.
6. Multiple account holders accessing their accounts simultaneously on the server. When you insert an ATM card, it starts a thread to perform your operations.
7. Encrypting files on a background thread, while an application runs on the main thread.

Merge Function

In the merging function, we use three while loops. The first one is to iterate over the two parts together. In each step, we take the smaller value from both parts and store it inside the temp array that will hold the final answer.

Once we add the value to the resulting temp, we move the index one step forward. The variable index points to the index that should hold the next value to be added to temp.

In the second while loop, we iterate over the remaining elements from the first part. We store each value inside temp. In the third while loop, we perform a similar operation to the second while loop. However, here we iterate over the remaining elements from the second part.

The second and third while loops are because after the first while loop ends, we might have remaining elements in one of the parts. Since all of these values are larger than the added ones, we should add them to the resulting answer.

The complexity of the merge function is O(len1 + len2) , where len1 is the length of the first part, and len2 is the length of the second one. Note that the complexity of this function is linear in terms of the length of the passed parts. However, it's not linear compared to the full array A because we might call the function to handle a small part of it.

Algorithm 1: Merge Function

Data: A: The array to be sorted
L1: The start of the first part
R1: The end of the first part
L2: The start of the second part
R2: The end of the second part
Result: Return the merged sorted array

Function $merge(A, L1, R1, L2, R2)$**:**
    $temp \leftarrow \{\}$;
    $index \leftarrow 0$;
    while $L1 \leq R1$ ***AND*** $L2 \leq R2$ **do**
        if $A[L1] \leq A[L2]$ **then**
            $temp[index] \leftarrow A[L1]$;
            $index \leftarrow index + 1$;
            $L1 \leftarrow L1 + 1$;
        else
            $temp[index] \leftarrow A[L2]$;
            $index \leftarrow index + 1$;
            $L2 \leftarrow L2 + 1$;
        end
    end
    while $L1 \leq R1$ **do**
        $temp[index] \leftarrow A[L1]$;
        $index \leftarrow index + 1$;
        $L1 \leftarrow L1 + 1$;
    end
    while $L2 \leq R2$ **do**
        $temp[index] \leftarrow A[L2]$;
        $index \leftarrow index + 1$;
        $L2 \leftarrow L2 + 1$;
    end
    return $temp$;
end

Merge Sort

Firstly, we start len from 1 which indicates the size of each part the algorithm handles at this step. In each step, we iterate over all parts of size len and calculated the beginning and end of each two adjacent parts. Once we determined both parts, we merged them using the merge function defined in algorithm 1.

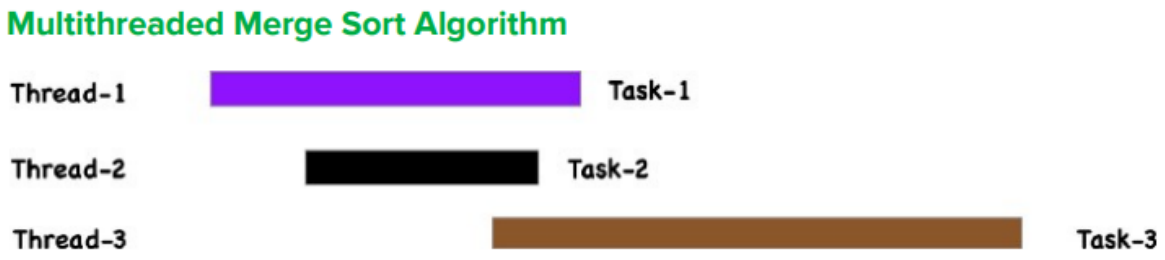
Note that we handled two special cases. First one is if L2 reaches the outside of the array, while the second one when R2 reaches the outside. The reason for these cases is that the last part may contain fewer than len elements. Therefore we adjust its size so that it doesn't exceed n.

After the merging ends, we copy the elements from temp into their respective places in A.
Note that in each step, we doubled the length of a single part len. The reason is that we merged two parts of length len. So, for the next step, we know that all parts of the size are now sorted. Finally, we return the sorted A.

The complexity of the iterative approach is *O(n * log(n))* , where n is the length of the array. The reason is that, in the first while loop, we double the value len in each step. So, this is *O(log(n)).* Also, in each step, we iterate over each element inside the array twice and call the merge function for the complete array total. Thus, this is O(n).

Algorithm 2: Iterative Merge Sort

```
Data: A: The array
      n : The size of the array
Result: Returns the sorted array
len ← 1;
while len < n do
    i ← 0;
    while i < n do
        L1 ← i;
        R1 ← i + len − 1;
        L2 ← i + len;
        R2 ← i + 2 × len − 1;
        if L2 ≥ n then
            break;
        end
        if R2 ≥ n then
            R2 ← n − 1;
        end
        temp ← merge(A, L1, R1, L2, R2);
        for j ← 0 to R2 − L1 + 1 do
            A[i + j] ← temp[j];
        end
        i ← i + 2 × len;
    end
    len ← 2 × len;
end
return A;
```



We employ divide-and-conquer algorithms for parallelism because they divide the problem into independent subproblems that can be addressed individually. Let's take a look at merge sort:

Algorithm 6: Merge Sort (parallel)

```
Merge-Sort(a, p, r)
if p < r then
    q = (p + r)/2
    spawn Merge-Sort(a, p, q)
    Merge-Sort(a, q+1, r)
    sync
    Merge(a, p, q, r)
```

As we can see, the dividing is in the main procedure Merge-Sort, then we parallelize it by using spawn on the first recursive call. Merge remains a serial algorithm, so its work and span are as before.

Modules

Multiprocessing -

Multiprocessing is a package that supports spawning processes using an API similar to the threading module. The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads.

Timeit -

Timeit module provides a simple way to time small bits of Python code. It avoids a number of common traps for measuring execution times.

Complexity Analysis

The dividing is in the main procedure Merge-Sort, then we parallelize it by using spawn on the first recursive call. Merge remains a serial algorithm, so its work and span are O(n) as before. Here's the recurrence for the work T1(n) of Merge-Sort (it's the same as the serial version):

$$T_1(n) = 2 \times T_1(\frac{n}{2}) + \theta(n) = \theta(n \times \log(n))$$

The recurrence for the span of Merge-Sort is based on the fact that the recursive calls run in parallel:

$$T_\infty(n) = T_\infty(\frac{n}{2}) + \theta(n) = \theta(n)$$

Here's the parallelism:

$$\frac{T_1(n)}{T_\infty(n)} = \theta(\frac{n \times \log(n)}{n}) = \theta(\log(n))$$

As we can see, this is low parallelism, which means that even with massive input, having hundreds of processors would not be beneficial. So, to increase the parallelism, we can speed up the serial Merge.

Code :

```
import argparse
import random
from contextlib import contextmanager
from multiprocessing import Pool, cpu_count
from timeit import default_timer as time

from tabulate import tabulate

CPU_COUNT = cpu_count()


class Timer:
  """
  Record timing information.
  """

  def __init__(self, *steps):
    self._time_per_step = dict.fromkeys(steps)

  def __getitem__(self, item):
    return self.time_per_step[item]

  @property
  def time_per_step(self):
    return {
      step: elapsed_time
      for step, elapsed_time in self._time_per_step.items()
      if elapsed_time is not None and elapsed_time > 0
    }

  def start_for(self, step):
    self._time_per_step[step] = -time()
```

```
    def stop_for(self, step):
        self._time_per_step[step] += time()


def merge_sort(array):
    """Perform merge sort."""
    array_length = len(array)

    if array_length <= 1:
        return array

    middle_index = array_length // 2
    left = merge_sort(array[:middle_index])
    right = merge_sort(array[middle_index:])
    return merge(left, right)


def merge(*arrays):
    """Merge two sorted lists."""

    # Support explicit left/right args, as well as a two-item
    # tuple which works more cleanly with multiprocessing.
    left, right = arrays[0] if len(arrays) == 1 else arrays
    sorted_list = [0] * (len(left) + len(right))
    i = j = k = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_list[k] = left[i]
            i += 1
        else:
            sorted_list[k] = right[j]
            j += 1
        k += 1

    while i < len(left):
        sorted_list[k] = left[i]
        i += 1
        k += 1
```

```
    while j < len(right):
        sorted_list[k] = right[j]
        j += 1
        k += 1

    return sorted_list

def parallel_merge_sort(array, ps_count):
    """Perform parallel merge sort."""
    timer = Timer("sort", "merge", "total")
    timer.start_for("total")
    timer.start_for("sort")

    # Divide the list in chunks
    step = int(len(array) / ps_count)

    # Creates a pool of worker processes, one per CPU core.
    # We then split the initial data into partitions, sized equally per
    # worker, and perform a regular merge sort across each partition.
    with process_pool(size=ps_count) as pool:
        array = [array[i * step : (i + 1) * step] for i in range(ps_count)] + [array[ps_count * step :]]
        array = pool.map(merge_sort, array)

        timer.stop_for("sort")
        timer.start_for("merge")

        # We can use multiprocessing again to merge sub-lists in parallel.
        while len(array) > 1:
            extra = array.pop() if len(array) % 2 == 1 else None
            array = [(array[i], array[i + 1]) for i in range(0, len(array), 2)]
            array = pool.map(merge, array) + ([extra] if extra else [])

        timer.stop_for("merge")
        timer.stop_for("total")

    final_sorted_list = array[0]

    return timer, final_sorted_list
```

Results

It was observed that the multi-threaded approach always had a better performance time than the Single Threaded approach. The difference in their performance started becoming visible as the input sizes began growing.

The following data has been calculated by varying the length of the array while keeping the other parameters constant with the values:- Number of Multiprocessing cores = 16

Array Length	Single-Threaded	Multi-Threaded
1	2.934e-06	0.0285377
10	2.2908e-05	0.0246235
100	0.000198492	0.0248281
1000	0.00220744	0.0421945
10000	0.0243417	0.0359442
100000	0.297823	0.166326
1000000	3.68507	1.23563
10000000	45.5687	11.9969

Table 1. Result of the program on different parameters of Input in seconds

Conclusion

In this project we have successfully implemented merge sort by using a multithreading approach. We have also analyzed the performance and complexity of merge sort using a multithreading approach. It was observed that the multi-threaded approach performs better than the naive approach by a great margin.