AISSMS
INSTITUTE OF INFORMATION TECHNOLOGY
(IOIT)
ADDING VALUE TO ENGINEERING
An Autonomous Institute Affiliated to Savitribai Phule Pune University
Approved by AICTE, New Delhi and Recognised by Govt. of Maharashtra
Accredited by NAAC with "A+" Grade | NBA - 5 UG Programmes

2023-2024

Department of Computer Engineering
MINI PROJECT REPORT ON

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

Submitted By

76	Atharva Mohan Tirkhunde	72145918J

Guided by
Ms. Shilpa Pimpalkar

AIM

Implement merge sort and multithreaded merge sort.

PROBLEM STATEMENT

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

SYSTEM REQUIREMENT

Operating System: 64-bit Linux or its derivatives / Windows.
Python Programming Language >= 3.1
Multiprocessing Module >= 2.6
Timeit Module >= 3.

OBJECTIVE

• Implement merge sort using multi-threading.
• To analyse the performance of the multithreading approach on merge sort.
• To analyse the complexity performance of the multithreading 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.

INTRODUCTION:

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.

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 ← {};
    index ← 0;
    while L1 ≤ R1 AND L2 ≤ R2 do
        if A[L1] ≤ A[L2] then
            temp[index] ← A[L1];
            index ← index + 1;
            L1 ← L1 + 1;
        else
            temp[index] ← A[L2];
            index ← index + 1;
            L2 ← L2 + 1;
        end
    end
    while L1 ≤ R1 do
        temp[index] ← A[L1];
        index ← index + 1;
        L1 ← L1 + 1;
    end
    while L2 ≤ R2 do
        temp[index] ← A[L2];
        index ← index + 1;
        L2 ← 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. The first one is if *L2* reaches the outside of the array, while the second one is 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 $2 \times len$ are now sorted.

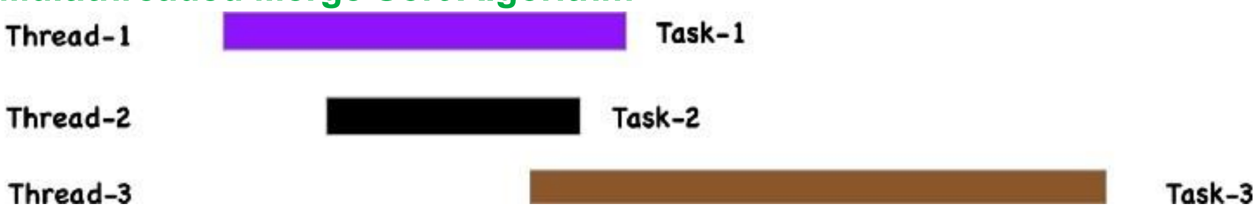
Finally, we return the sorted *A*.

The complexity of the iterative approach is $\boldsymbol{O(n \times log(n))}$, 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
$len \leftarrow 1$;
while $len < n$ **do**
  $i \leftarrow 0$;
  while $i < n$ **do**
    $L1 \leftarrow i$;
    $R1 \leftarrow i + len - 1$;
    $L2 \leftarrow i + len$;
    $R2 \leftarrow i + 2 \times len - 1$;
    if $L2 \geq n$ **then**
      break;
    end
    if $R2 \geq n$ **then**
      $R2 \leftarrow n - 1$;
    end
    $temp \leftarrow merge(A, L1, R1, L2, R2)$;
    for $j \leftarrow 0$ **to** $R2 - L1 + 1$ **do**
      $A[i + j] \leftarrow temp[j]$;
    end
    $i \leftarrow i + 2 \times len$;
  end
  $len \leftarrow 2 \times len$;
end
return A;

Multithreaded Merge Sort Algorithm



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)$_{\overline{2}}$
  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 $\theta(n)$ as before.

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.

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 $\theta(n)$ as before.
Here’s the recurrence for the work $T_1(n)$ 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 $T_\infty(n)$ 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:

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

OUTCOME

- A multi-threaded approach to *Merge Sort* was implemented and analyzed in accordance with the main thread naive approach.
- The results prove that the multi-threaded approach performs better than the naive approach by a great margin.
- This minor optimization has led to its many industrial applications like ATM transactions, ticket reservations, etc. to name a few.

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.

SCREENSHOTS

```
$> python3 "merge_sort.py" -j 16
Using 16 cores

List length: 3332517
Random list generated in 1.826744s

Starting simple sort.
Single Core elapsed time: 14.580756s

Starting parallel sort.
Final merge duration: 2.910911s
16-Core elapsed time: 5.131196s
----------------------------------------
+----------------+-------------------+------------------+
|   Array Length |   Single-Threaded |   Multi-Threaded |
+================+===================+==================+
|        3332517 |           14.5808 |           5.1312 |
+----------------+-------------------+------------------+

$> python3 "merge_sort.py" -a
Using 16 cores

List length: 1
Random list generated in 0.000005s

Starting simple sort.
Single Core elapsed time: 0.000003s

Starting parallel sort.
Final merge duration: 0.003702s
16-Core elapsed time: 0.028538s
----------------------------------------

List length: 10
Random list generated in 0.000035s

Starting simple sort.
Single Core elapsed time: 0.000023s

Starting parallel sort.
Final merge duration: 0.002112s
16-Core elapsed time: 0.024624s
----------------------------------------

List length: 100
Random list generated in 0.000125s

Starting simple sort.
Single Core elapsed time: 0.000198s
```

```
Starting parallel sort.
Final merge duration: 0.001935s
16-Core elapsed time: 0.024828s
----------------------------------------

List length: 1000
Random list generated in 0.000616s

Starting simple sort.
Single Core elapsed time: 0.002207s

Starting parallel sort.
Final merge duration: 0.004501s
16-Core elapsed time: 0.042195s
----------------------------------------

List length: 10000
Random list generated in 0.006530s

Starting simple sort.
Single Core elapsed time: 0.024342s

Starting parallel sort.
Final merge duration: 0.009236s
16-Core elapsed time: 0.035944s
----------------------------------------

List length: 100000
Random list generated in 0.061283s

Starting simple sort.
Single Core elapsed time: 0.297823s

Starting parallel sort.
Final merge duration: 0.081855s
16-Core elapsed time: 0.166326s
----------------------------------------

List length: 1000000
Random list generated in 0.612833s

Starting simple sort.
Single Core elapsed time: 3.685065s

Starting parallel sort.
Final merge duration: 0.645426s
16-Core elapsed time: 1.235625s
----------------------------------------

List length: 10000000
Random list generated in 5.445338s

Starting simple sort.
Single Core elapsed time: 45.568661s
```

```
Starting parallel sort.
Final merge duration: 5.418382s
16-Core elapsed time: 11.996948s
----------------------------------------
+----------------+-------------------+------------------+
|   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 |
+----------------+-------------------+------------------+
```

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


@contextmanager def
process_pool(size):
    """Create a process pool and block until all processes have completed.""" pool = Pool(size)
    yield pool pool.close() pool.join()


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:
        # If the number of partitions remaining is odd, we pop off the
        # last one and append it back after one iteration of this loop, # since we're only
        interested in pairs of partitions to merge.
        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


def get_command_line_parameters():
    """Get the process count, array length from command line parameters."""

    parser = argparse.ArgumentParser( description="""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."""
    ) parser.add_argument(
        "-j", "--jobs", help="Number of processes to
        launch", required=False,
        default=CPU_COUNT,
```

```
            type=lambda x: int(x) if 0 < int(x) <= CPU_COUNT else
            parser.error(f"Number of processes must be between 1 and
{CPU_COUNT}"),
      ) parser.add_argument(
            "-l", "--length", help="Length of the array to sort", required=False,
            default=random.randint(3 * 10**6, 4 * 10**6), # Randomize the length of
our list type=lambda x: int(x) if 0 < int(x) else parser.error("Length of the array
must be greater than 0"),
      ) parser.add_argument(
            "-a", "--all", help="Test all the variable
            length", required=False, default=False,
            action="store_true",
      ) return parser.parse_args()


def main(jobs, length, conclusion):
      """Main function."""

      main_timer = Timer("single_core", "list_generation") main_timer.start_for("list_generation")

      # Create an unsorted list with random numbers randomized_array = [random.randint(0,
      i * 100) for i in range(length)] main_timer.stop_for("list_generation")

      print(f"List length: {length}") print(f"Random list generated in
      {main_timer['list_generation']:.6f}s\n")

      # Create a copy first due to mutation randomized_array_sorted =
      randomized_array[:] randomized_array_sorted.sort()

      # Start timing the single-core procedure print("Starting
      simple sort.") main_timer.start_for("single_core")
      sorted_array = merge_sort(randomized_array)
      main_timer.stop_for("single_core")
```

```
        # Comparison with Python list sort method # serves also as
        validation of our implementation.
        assert sorted_array == randomized_array_sorted, "The sorted array is not
correct." print(f"Single Core elapsed time: {main_timer['single_core']:.6f}s\n")

        print("Starting parallel sort.") parallel_timer, parallel_sorted_array =
        parallel_merge_sort(randomized_array,
jobs) print(f"Final merge duration: {parallel_timer['merge']:.6f}s")

        assert parallel_sorted_array == randomized_array_sorted, "The sorted array is
not correct." print(f"{jobs}-Core elapsed time: {parallel_timer['total']:.6f}s\n" + "-" * 40,
"\n") conclusion.append([length, main_timer["single_core"], parallel_timer["total"]])


if __name__ == "__main__": parameters =
        get_command_line_parameters()

        jobs = parameters.jobs length =
        parameters.length all_cases =
        parameters.all conclusion = []
        print(f"Using {jobs} cores\n")

        if all_cases:
                l = 1 while l < 10**8: main(jobs, l,
                conclusion) l *= 10
        else:
                main(jobs, length, conclusion)

        print(tabulate(conclusion, headers=["Array Length", "Single-Threaded", "Multi-Threaded"],
tablefmt="outline"))
```