CEG5201 Hardware Technologies, Principles, & Platforms
(Semester I, AY2024/2025)
CA-2 Statement

Release date: 25th – 27th Sept 2024
Consultation date: 16th Oct 2024
Submission (Hard) deadline: 13th Nov 2024 11:59 pm

Instructions:

The CA2 assignment must be submitted on Canvas by the **deadline stated above**. Note that **late submissions will not be accepted**, so avoid last-minute submissions to prevent technical issues.

I. Objective

In this group assignment, each member will focus on benchmarking one of the following sorting algorithms—**Merge Sort**, **Bucket Sort**, **Quicksort**, or **Odd-Even Transposition Sort**—using both **sequential** and **multiprocessing** methods. The objective is to compare the performance of these algorithms across different implementations.

To complete the assignment, all group members should familiarize themselves with the Python `multiprocessing` package using the following resources:

- Python multiprocessing tutorial: https://www.datacamp.com/tutorial/python-multiprocessing-tutorial
- The pool class - https://superfastpython.com/multiprocessing-pool-python/
- Multiprocessing pool example - https://superfastpython.com/multiprocessing-pool-example/
- Threadpool vs. pool class differences - https://superfastpython.com/threadpool-vs-pool-in-python

Feel free to explore additional learning materials and share useful links by posting on the Canvas Discussion to benefit your classmate

II. Project Overview

In this project, your team will implement four sorting algorithms shown in Table II. Each team will select one of the sorting algorithms and implement it in both sequential and multiprocessing modes. Furthermore, in addition to individual tasks, the project includes joint tasks as detailed in Table I.

- **Individual tasks** are highlighted as blue.
- **Joint tasks** are highlighted as yellow.

Table I. Overview of the tasks

Task ID	Task description
O	**Getting started**
$\boldsymbol{A}$	Unsorted array generation
$\boldsymbol{B_n}$	Sequential implementation
$\boldsymbol{C_n}$	Multiprocessing implementation
$\boldsymbol{D_n}$	Determining the ultimate speed-up
$\boldsymbol{E}$	Comparison of the algorithms

For tasks B, C, and D, each team member will work on **one algorithm**. The algorithm assigned to each member is determined using the following formula:

$$\boldsymbol{n = (groupID + memberID)\%\ 4 + 1}$$

Where **%** shows the modulo operation and the algorithm corresponding to $\boldsymbol{n}$ is listed in Table II.

Table II. List of the algorithms used

S/N (n)	Algorithm
1	Merge sort
2	Bucket sort
3	Quicksort
4	Odd-even transposition sort

Refer to **Appendix A** for more details on each algorithm.

III. Task description

[O] Getting started

As a team, first fix the hardware platform that you are going to use – multi-core CPU / GPU before doing individual coding.

[A] Unsorted Array Generation

Generate **8 unsorted** array instances, denoted as $\boldsymbol{A_i}$ where $i \in [0, 1, \ldots 7]$. The length denoted by $\boldsymbol{N_i}$ of each array instance $\boldsymbol{A_i}$ is chosen from the set {64, 128, 256, 512, 1024, 2048, 4096, 8192}. Note that the value of each element $A_i[j]$, should be between **0 and 255** (*random* integers only). Next, generate 10 groups of such list of arrays G_j, $j \in [0, 1, \ldots 9]$. **All the team members need to use the same set of G for their subsequent tasks.**

[B_n] Sequential Processing

For **[B_{n1}]** follow these instructions:

- Sort all 8 arrays in group $\boldsymbol{G_0}$ sequentially.

- Record the time taken for each array $\boldsymbol{A_i}$ and the cumulative total time. This will give you the sequential processing time for the group G_0.
- Present your results in Table $\boldsymbol{B_{n1}}$ as shown below.
- Put the table in your report exactly as it appears.

Table $\boldsymbol{B_{n1}}$: Processing time of G_0 under sequential implementation

Array $\boldsymbol{A_i}$	Measured Sequential Time	Cumulative Sequential time
0	t_0	t_0
1	t_1	$t_0 + t_1$
2	t_2	$t_0 + t_1 + t_2$
…	…	…
7	t_7	$t_0 + t_1 + \cdots + t_7$

For **[$\boldsymbol{B_{n2}}$]** follow these instructions:

- Repeat the above processes for all the 10 groups. This is the sequential processing time of all groups
- Present the results obtained in Table $\boldsymbol{B_{n2}}$ shown below. Please put the table in your report exactly as it appears.

Table $\boldsymbol{B_{n2}}$: Processing time of all groups under sequential implementation

Group Index	Sequential Time (Group)	Cumulative Sequential time (Group)
0	t_0	t_0
1	t_1	$t_0 + t_1$
2	t_2	$t_0 + t_1 + t_2$
…	…	…
9	t_9	$t_0 + t_1 + \cdots + t_9$

[C_n] Multiprocessing

Identify the bottleneck in the sequential processing and try to accelerate it using multiple processes.

For **[$\boldsymbol{C_{n1}}$]** follow these instructions:

- Record the time it takes to complete the processing of $\boldsymbol{G_0}$ by varying the number of processes.
- Present the obtained results in Table $\boldsymbol{C_{n1}}$ shown below. Note, the columns with number of processes $\geq$ 8, are optional for you to fill.
- Put the table in your report exactly as it appears.

Table C_{n1}: Processing time of G_0 under multiprocessing implementation

	Measured MP time						Measured Cumulative MP Time					
Process → Array A_i↓	1	2	4	8	…	…	1	2	4	8	…	…
0												
1												
2												
3												
…												
7												

For **[C_{n2}]** follow these instructions:

- Record the time it takes to complete the processing of all groups by varying the number of processes.
- Present the obtained results in Table C_{n2} shown below. Note the columns with number of processes $\geq$ 8, are optional for you to fill.
- Put the table in your report exactly as it appears.

Table C_{n2}: Processing time of all groups under multiprocessing implementation

	Measured MP time						Measured Cumulative MP Time					
Process → Grp Index↓	1	2	4	8	…	…	1	2	4	8	…	…
0												
1												
2												
3												
…												
9												

[D_n] Determine the speed-up

For **[D_{n1}]** follow these instructions:

- Determine the speed-up of processing G_0 under multiprocessing implementation (C_{n1}) versus sequential implementation (B_{n1}).

For **[D_{n2}]** follow these instructions:

- Determine the speed-up of processing all groups under multiprocessing implementation (C_{n2}) versus sequential implementation (B_{n2}).
- Plot the speed-up curves w.r.t the number of processes and the number of workloads (pairs/groups) and write a detailed discussion.

[E] Compare the algorithms

Combine the results from all team members and provide a detailed comparison of the different algorithms in your report.

IV. Submission

What to submit?

Each group should upload **only one compressed file (.zip)** to *Canvas*, including three types of files:

1. **The report (.pdf)**
2. **The source code file(s) (.py)**
3. **The README text file (.txt)**

File Naming Instructions

Follow the rules below to name your files:

- For the compressed file:
  - ✓ CA2_(Group ID)_(Matriculation number of all team members separated by an underscore).zip
  - ✓ Example: CA2_12_A0213331Z_A0010101Y_A0340127X_A0711001X.zip
- For the report file:
  - ✓ CA2_Report_(Group ID)_(Matriculation number of all team members separated by an underscore).pdf
  - ✓ Example:CA2_Report_12_A0213331Z_A0010101Y_A0340127X_A0711001X.pdf

Report Format

- **Formatting Guidelines**:
  Use **single-column format** with the main body text in **Times New Roman**, **12pt** font.

- **Page Limit**:
  Maximum of **12 pages**, **excluding** the first and last pages.

- **First Page of your report**:
  Title + Full Names + Matriculation Numbers (this page is not counted in the report page limit mentioned below).

- **Last Page of your report:**
  Title: Coding effort (this page is not counted in the report page limit mentioned below).
  Please see the next subsection on what to write here.

- **Code file(s)**:
  Please put useful comments generously throughout in your code file for readers to understand the gist of computation taking place.

Coding effort

Attach ONE separate page titled "Coding Effort - <your name>" at the end of report. Describe how much your coding effort is (quantify in %). If you had either directly or partly used any CODE(s) from *github* or from any other site, list those sites and point us exactly where you obtained those codes. If you have used others' codes and did not list or cite the reference directly no marks will be awarded. If you have generated your own data, describe how you generated your data needed for your experiments. If you are using any tools for your experiments (not for data), then point to them. These also can be part of your references listing after conclusions.

Readme file

This file must contain clear instructions to facilitate running your program and use of data and parameters. If you are using any specific packages, list them and also indicate the ***URLs*** from where the readers can download. If you are using any specific ***DATA*** from any URL, indicate the URL. If the readers need to set any input parameters (hard coded way) explicitly state that requirement. Do **NOT ASSUME** anything and omit details. Step-by-step instructions will be very useful.

V. Assessment

Grading

Your assignment will be **graded out of 60 marks**, and the final weight of this assignment is **35%.** For the joint effort, all members will be awarded identical marks. The mark breakdown is shown in Table III.

Table III. Grade breakdown

Joint effort (50%)	Effort in (O), (A) and (E). Title, abstract, introduction, joint discussions, and conclusion.
Individual effort (50%)	Effort in (B), (C), and (D). Sections on interpreting individual results.

Plagiarism Penalty

You are allowed to use only 15% of any written material collectively from all other sources. Anything more than this 15% will be penalized. References cited may show up in your plagiarism checker and you can ignore this percentage. **Copying of coding and simulation results directly from others will be awarded 0 marks for both Source and Copier(s)**.

Rubrics

Criteria	Remarks	Points
[Joint] Abstract	Capture the algorithms and implementation approaches used and highlight of any significant results in not more than 12 lines.	2
[Joint] Introduction	Brief introduction of the algorithms and implementation approaches. Clear problem statement in plain English and also using technical description.	5
[Joint]	Describe your input groups of unsorted arrays and how they are	6

Description of input data [A]	generated. If you are using any special packages, please indicate that clearly.	
[Indiv] Description of algorithm and implementation approach [B, C]	Each one of you can open a separate sub-section in this part. Describe the algorithms in a concise fashion. Highlight any non-trivial decisions/steps that the algorithm takes and use proper citations. You may use a simple numerical example to explain if you wish. Describe clearly how you implement the multi-processing method. Elaborate how you identify the bottleneck and maximize the parallelism and acceleration enabled by the multi-processing approach.	10
[Indiv] Individual results and discussions [B, C]	Each one of you can open a separate sub-section in this part. Put all your simulation results in a systematic fashion under each subsection, point to the results, and argue clearly about the trends, behaviour, results, etc. Clearly interpret and discuss your table/graphs and try to relate to the algorithms described in the earlier section.	15
[Joint] Joint discussions [E]	This is the joint discussion under Task E. If your arguments/discussions/comments are valid, you may open a common section and compare the algorithms. This demonstrates your group effort. Describe the strengths and pitfalls of the algorithms, if any, clearly. Argue why algo(s) fails in certain conditions and what it does not consider, etc. You may describe here what you feel is right and wrong that is within the scope of formulations used in the paper.	12
[Joint] Conclusions & reference(s) used [E]	Conclude how the algorithms benefit/suffer from the multi-processing and the potentials/limitations to the number of processes/workloads in not more than 12 lines. Formally list the paper(s) & sites you have directly used.	5
[Indiv] Coding effort	Report here your coding effort.	5
TOTAL MARKS		**60**

Appendix A

Below is a general overview of the four sorting algorithms. The parallel versions provided are for reference only; you may implement parallelization in your own way if preferred.

1. Merge Sort

Merge sort is a classic divide-and-conquer sorting algorithm [1]. It recursively splits the array into smaller subarrays until each subarray contains a single element, and then merges the subarrays back together in sorted order. Although merge sort has a time complexity of $\mathbf{O}(\boldsymbol{n}.\mathbf{log}\boldsymbol{n})$, it is inherently recursive and can be parallelized, offering opportunities to speed up the process using multiple threads or processors [2].

Here we show the pseudocode of serial merge sort algorithm:

```
function mergeSort(arr):
    if length(arr) <= 1:
        return arr

    mid = length(arr) / 2
    left = mergeSort(arr[0:mid])
    right = mergeSort(arr[mid:length(arr)])

    return merge(left, right)

function merge(left, right):
    result = []
    while left is not empty and right is not empty:
        if left[0] <= right[0]:
            append left[0] to result
            remove left[0]
        else:
            append right[0] to result
            remove right[0]

    append any remaining elements in left to result
    append any remaining elements in right to result

    return result
```

And pseudocode of parallel version:

```
function mergeSort(arr):
    if length(arr) <= 1:
        return arr

    mid = length(arr) / 2
    left = mergeSort(arr[0:mid])
    right = mergeSort(arr[mid:length(arr)])

    return merge(left, right)

function merge(left, right):
    result = []
    while left is not empty and right is not empty:
        if left[0] <= right[0]:
            append left[0] to result
            remove left[0]
        else:
            append right[0] to result
            remove right[0]

    append any remaining elements in left to result
    append any remaining elements in right to result

    return result
```

Note that you can use a "depth" parameter to limit the level of parallelism, where parallel processing only happens at a certain recursion depth.

2. Bucket Sort

A sequential version of bucket sort is described in [3]:

```
function BucketSort(arr):

    if length(arr) <= 1:
        return arr

    # Create n empty buckets
    n = length(arr)
    buckets = [] for _ in range(n)

    # Put elements into respective buckets
```

```
    for element in arr:
        index = floor(n * element)
        append element to buckets[index]

    # Sort each bucket
    for bucket in buckets:
        bucket = Sort(bucket)  # Use any sorting algorithm

    # Concatenate all sorted buckets
    result = []
    for bucket in buckets:
        result.extend(bucket)

    return result
```

The parallel version of bucket sort is introduced as well:

```
function BucketSort(arr):

    if length(arr) <= 1:
        return arr

    # Create n empty buckets
    n = length(arr)
    buckets = an array of n empty lists

    # Distribute elements into buckets
    for each element in arr:
        index = floor(n * element)
        insert element into buckets[index]

    # Sort each bucket in parallel
    sorted_buckets = ParallelMap(Sort, buckets)  # Use any sorting
algorithm

    # Concatenate sorted buckets
    result = an empty list
    for each bucket in sorted_buckets:
        append all elements of bucket to result

    return result
```

Refer to [3] for performance comparison.

3. Quicksort

The pseudocode of sequential quicksort is listed below:

```
algorithm quicksort(A, lo, hi)
    if lo >= 0 and hi >= 0 and lo < hi then
        p = partition(A, lo, hi)
        quicksort(A, lo, p)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[lo]
    i := lo - 1
    j := hi + 1

    loop forever
        do i := i + 1 while A[i] < pivot
        do j := j - 1 while A[j] > pivot
        if i >= j then return j
        swap A[i] with A[j]
```

For the parallel version of quicksort, a possible solution is provided below. First assume we have p processes and each is distributed part of the unsorted array. A pivot is randomly chosen from one of the processes and is broadcast to every process. Each process divides its unsorted list into two lists: those smaller than (or equal) the pivot, those greater than the pivot. Each process in the upper half of the process list sends its "low list" to a partner process in the lower half of the process list and receives a "high list" in return. Now, the upper-half processes have only values greater than the pivot, and the lower-half processes have only values smaller than the pivot. Thereafter, the processes divide themselves into two groups and the algorithm recurses. After log(p) recursions, every process has an unsorted list of values **completely disjoint** from the values held by the other processes. Then we can apply sequential quicksort on each process.

Note the aforementioned algorithm can suffer from poor load balancing because of the **choice of the pivot**. There are some improvements can be done. For more details, check Chapter 14 of [4].

4. Odd-Even Transposition Sort

Odd-even transposition sort (a.k.a odd-even sort or brick sort) is similar to bubble sort but is designed for parallelism [5]. The compare-swap operations are separated into two phases: in **even phase**, compare-swap are only performed on pairs $(a_0, a_1), (a_2, a_3), (a_4, a_5), \ldots$ while in **odd phase**, compare-swap are only performed on pairs $(a_1, a_2), (a_3, a_4), (a_5, a_6), \ldots$ The pseudocode code below illustrates the sequential odd-even transposition sort.

```python
def odd_even_sort(arr: list[int]):
    n = len(arr)
    for i in range(n):
        # even phase
        if i % 2 == 0:
            for j in even_numbers:
                if arr[j] > arr[j + 1]:
                    swap(arr, j, j + 1)
        # odd phase
        else:
            for j in odd_numbers:
                if arr[j] > arr[j + 1]:
                    swap(arr, j, j + 1)
```

Note the pseudocode above is not optimal, as the algorithm can be terminated earlier if no compare-swap operations are performed in one iteration of the outer loop (i.e. the array has been sorted.)

For the parallel version of odd-even transposition sort, one possible solution is in either phase, the compare-swap operations can be performed simultaneously. For instance, compare-swap of (a_0, a_1) and (a_2, a_3) can be done parallelly without any side effects. For more details, check [5], [6] (Chapter 3.7 & 5.6).

References [*If you wish to do any additional reading you may use these references*]

[1] Goodrich, Michael T.; Tamassia, Roberto; Goldwasser, Michael H. (2013). "Chapter 12 - Sorting and Selection". Data structures and algorithms in Python (1st ed.). Hoboken [NJ]: Wiley. pp. 538–549. ISBN 978-1-118-29027-9.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022).“26.3 Parallel merge sort”. *Introduction to algorithms*. pp. 1004-1014. MIT press.

[3] H. Hong, “PARALLEL BUCKET SORTING ALGORITHM,” 2014. Accessed: Sep. 18, 2024. [Online]. Available: https://www.sjsu.edu/people/robert.chun/courses/cs159/s3/N.pdf.

[4] M. J. Quinn, Parallel Programming in C with MPI and OpenMP. pp. 340-349. McGraw-Hill Education Group, 2003.

[5] P. S. Pacheco, “Chapter 3 - Distributed-Memory Programming with MPI,” in *An Introduction to Parallel Programming*, P. S. Pacheco, Ed., Boston: Morgan Kaufmann, 2011, pp. 127–136. doi: 10.1016/B978-0-12-374260-5.00003-8.

[6] P. S. Pacheco, “Chapter 5 - Shared-Memory Programming with OpenMP,” in *An Introduction to Parallel Programming*, P. S. Pacheco, Ed., Boston: Morgan Kaufmann, 2011, pp. 232–236. doi: 10.1016/B978-0-12-374260-5.00005-1.

----- END -----