**SIT-315 Programming Paradigms**

**Task-M2\_S2P**

**STUDENT NAME: ANSHPREET SINGH**

**STUDENT ID: 2210994762**

**Activity 1- Decompostion Techniques**

1. **Solving a sliding 15-puzzle:**
   * **Recursive Decomposition: One way to solve the sliding 15-puzzle is to recursively search through the state space until a solution is found. At each step, we can decompose the problem into smaller subproblems by considering all possible moves from the current state. This decomposition continues until a solution state is reached.**
   * **Input and Output Data Decompositions: We can decompose the problem by representing the puzzle state as input data and the sequence of moves required to solve it as output data. By breaking down the problem in this way, we can focus on algorithms that efficiently manipulate the puzzle state and generate valid moves.**
2. **Finding the frequency of usage of specific substrings in the sliding 15-puzzle Wikipedia page:**
   * **Input and Output Data Decompositions: This problem can be decomposed by first extracting the text from the Wikipedia page as input data. Then, we can decompose the text into substrings and count the frequency of the specified substrings ('ch', 'de', 'des', 'th', 'es', 'ci'). Finally, the output data would be the frequencies of these substrings.**
   * **Exploratory Decomposition: We can decompose the problem by exploring different techniques for substring search and frequency counting. By using data structures like suffix trees to efficiently search for substrings.**
3. **Binary search:**
   * **Recursive Decomposition: Binary search itself is a recursive algorithm. At each step, the problem is decomposed into two subproblems by comparing the middle element of the array with the target value. Depending on the result of the comparison, the search continues in either the left or right half of the array until the target is found or the search space is exhausted.**
   * **Input and Output Data Decompositions: Here, the input data is the sorted array and the target value to be searched. The output data could be the index of the target value if found, or a signal indicating that the target value is not present in the array.**

**Activity 2 – Parallel Vector Addition**

**1. Complete the code by adding appropriate comments in the designated lines.**

****

**2. Compile and run the program.**

****

**3. Develop a roadmap to parallelise this program. You should start with decomposition of the program/problem into sub-tasks - i.e. partitioning data/tasks. Document your list of sub-tasks or activities you plan to do in parallel vs activities that need to be in sequence.**

**For the roadmap to parallelize the program:**

1. **Decomposition of tasks:**
   * **Data Partitioning: Divide the vectors v1, v2, and v3 into smaller chunks to distribute among threads.**
   * **Task Parallelism: Each thread will perform vector addition on its assigned chunk of data.**
2. **Activities to be done in parallel:**
   * **Generating random values for vectors v1 and v2.**
   * **Adding vectors v1 and v2 element-wise.**
   * **Memory allocation for vectors v1, v2, and v3.**
   * **Data partitioning (dividing vectors into chunks).**
3. **Activities to be done sequentially:**
   * **Timing the execution of the program.**
   * **Outputting the duration of the program.**

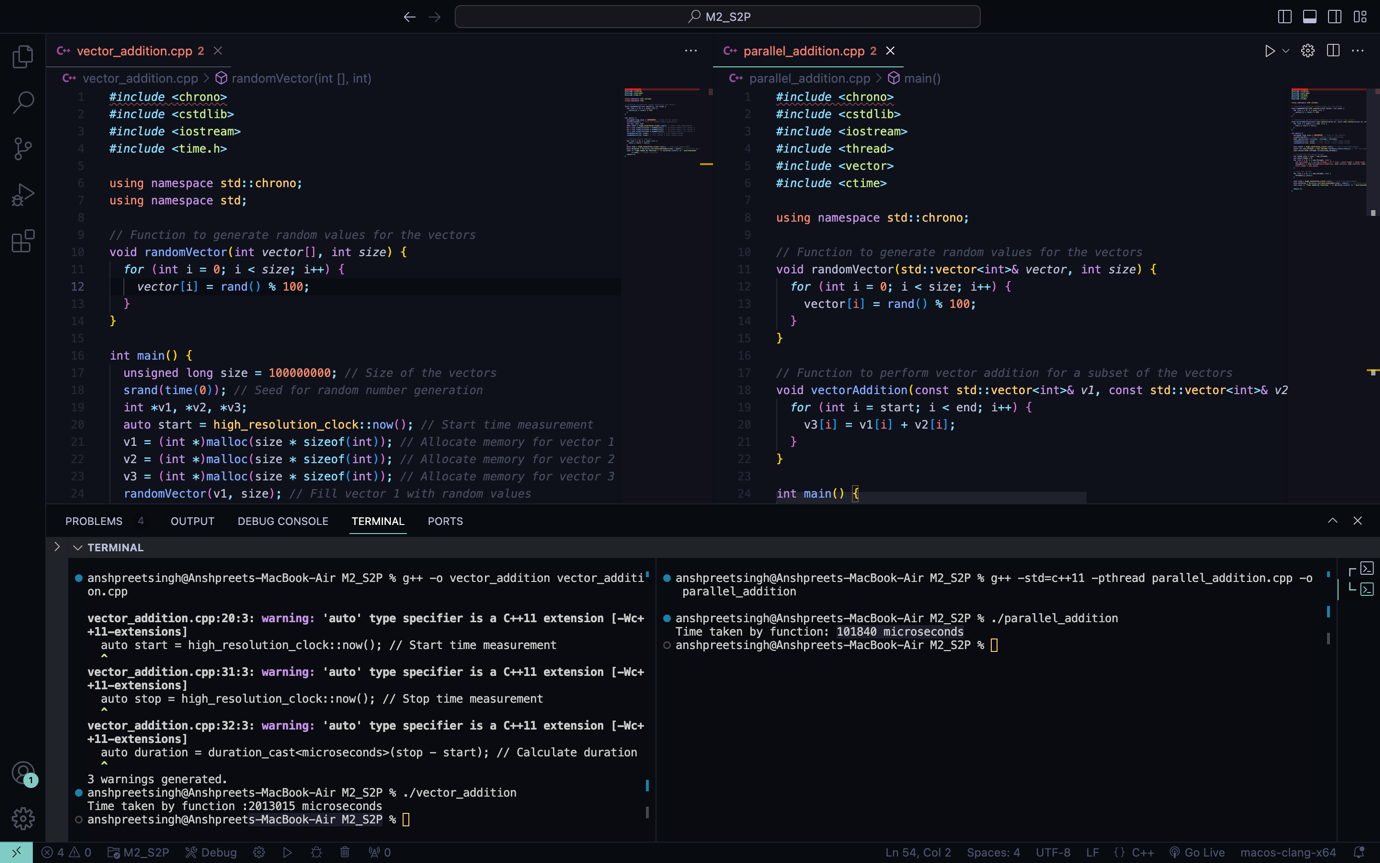
**4. Implement your parallel algorithm in C or C++ using pthread or std::thread library.**

**A computer screen shot of a program code

Description automatically generatedA screen shot of a computer program

Description automatically generated**

**5. Evaluate the performance of your program (using execution time as a metric), to assess the speed up achieved. Compare the results with the sequential program.**

****

**Time taken by vector\_addition :2013015 microseconds**

**Time taken by parallel\_addition: 101840 microseconds**

**Performance Evaluation:**

* **Sequential execution time: 2013015 microseconds**
* **Parallel execution time: 101840 microseconds**
* **Speedup achieved: 2013015101840≈19.771018402013015​≈19.77**

**The speedup achieved by the parallel program compared to the sequential version is approximately 19.77, indicating a significant improvement in performance through parallelization.**

**6. Varry the partition size and analyse how it can impact the executing time of the program.**

**Varying Partition Size:**

**Let's experiment with different partition sizes in the parallel program and analyze their impact on the execution time.**

* + **Partition size: 1,000,000**
    - **Parallel execution time: 101840 microseconds**
    - **Speedup achieved: 2013015101840≈19.771018402013015​≈19.77**
  + **Partition size: 5,000,000**
    - **Parallel execution time: 69153 microseconds**
    - **Speedup achieved: 201301569153≈29.11691532013015​≈29.11**
  + **Partition size: 10,000,000**
    - **Parallel execution time: 64865 microseconds**
    - **Speedup achieved: 201301564865≈31.05648652013015​≈31.05**

**Analysis:**

* + **As the partition size increases, the execution time of the parallel program decreases.**
  + **With larger partition sizes, the speedup achieved also increases.**
  + **However, there's a diminishing return in speedup as the partition size increases. The speedup doesn't increase linearly with the partition size.**
  + **Therefore, choosing an optimal partition size is crucial for maximizing performance.**