Hello! I would be happy to walk you through Merge Sort. This is one of the most important algorithms in computer science because it is efficient and stable.

We will start with the regular (sequential) version, and then look at the parallel version from your text.

Part 1: Sequential Merge Sort

Merge Sort uses a strategy called **Divide and Conquer**. It breaks a big problem into tiny problems, solves them, and puts them back together.

Sequential Pseudocode

Here is how a computer thinks about Merge Sort, one step at a time:

```
FUNCTION MergeSort(List):
    1. Check Size:
       If the List has 0 or 1 numbers,
return it. (It is already sorted!)

    2. Divide:
```

```
        Find the middle point to split the
List into two halves: LeftHalf and
RightHalf.

    3. Recurse (The Magic Step):
        SortedLeft = call
MergeSort(LeftHalf)
        SortedRight = call
MergeSort(RightHalf)
        *Note: The computer does the Left
side first, finishes it, THEN does the
Right side.*

    4. Conquer (Merge):
        Return Merge(SortedLeft,
SortedRight)
```

How it works (Step-by-Step Example)

Let's sort the list: **[4, 2, 3, 1]**

1. **Divide:** Split into **[4, 2]** and **[3, 1]**.
2. **Recurse (Left):** Look at **[4, 2]**. Split into **[4]** and **[2]**.

    - These are single numbers, so they are "sorted."
    - **Merge:** Put **[4]** and **[2]** together in order $\rightarrow$ **[2, 4]**.
3. **Recurse (Right):** Look at **[3, 1]**. Split into **[3]** and **[1]**.
    - **Merge:** Put **[3]** and **[1]** together in order $\rightarrow$ **[1, 3]**.
4. **Final Merge:** Take the two sorted halves **[2, 4]** and **[1, 3]**.
    - Compare 1 and 2 (1 is smaller).
    - Compare 2 and 3 (2 is smaller).
    - Compare 3 and 4 (3 is smaller).
    - 4 is left.
    - **Result: [1, 2, 3, 4]**.

Part 2: Parallel Merge Sort (From Figure 11)

Now, let's look at the code in your image (Figure 11). The goal here is speed. Instead of doing the Left

side and *then* the Right side, we want to do them **both at the same time**.

The Symbols in Figure 11

- A: The array (list) of numbers we want to sort.
- $|A|$: The length (size) of the array.
    - *Example:* If $A = [10, 20, 30]$, then $|A| = 3$.
- m: The middle index.
- $A[0 : m]$: A slice of the array from the start up to the middle.
- $||$ **(The Parallel Bar)**: This is the most important symbol. It means "Run the code on the left and the code on the right simultaneously."

Parallel Pseudocode Explanation

Here is the translation of the code in Figure 11:

```
MERGESORT(A):
    if (|A| = 1) then return A;      (Base
Case: If list has 1 item, stop.)
    else
       m = |A|/2                     (Find
the middle)
```

```
        l = MERGESORT(Left Half) || r =
MERGESORT(Right Half)
        (Do BOTH sorts at the exact same
time on different processors!)

        return MERGE(l, r)
(Combine the results)
```

Why is this faster? (Work vs. Span)

The text mentions two equations at the bottom of Section 5.1. Here is what they mean:

1. Work ($W(n)$):

- **Meaning:** The total number of operations (additions, comparisons) the computer performs.
- **Result:** Even though we are parallel, the *total* work is the same as the sequential version. We still have to compare every number.
- **Math:** $W(n) = 2W(n/2) + W_{\text{merge}}(n)$. (Two halves + merging cost).

2. Span ($D(n)$):

- **Meaning:** The "Depth" or "Critical Path." If you had infinite processors, how long would it take?
- **The Win:** Because the left and right sorts happen at the same time, we don't add their times together. We only count the time of *one* of them (the slower one, though they are usually equal).
- **Math:** $D(n) = D(n/2) + D_{\text{merge}}(n)$.
    - *Note:* In the Work equation, we multiplied by 2. Here, we do not. That is the power of parallelism!

Summary

- **Sequential:** Sort Left... wait... Sort Right... wait... Merge.
- **Parallel:** Spawn two helpers. Helper A sorts Left, Helper B sorts Right *simultaneously*. When they both finish, you Merge.

Quicksort

Here is the breakdown for Quick Sort. Just like before, we will start with the standard way a single

computer does it, and then see how your text speeds it up with parallel processing.

Part 1: Sequential Quick Sort

Quick Sort is another **Divide and Conquer** algorithm. However, it works differently than Merge Sort.

- **Merge Sort** splits the list right down the middle (by index).
- **Quick Sort** splits the list by **value**. It picks a number (called the **Pivot**) and throws everything smaller to the left and everything larger to the right.

Sequential Pseudocode

Here is how the logic works step-by-step:

```
FUNCTION QuickSort(List):
    1. Base Case:
       If the List is empty or has size 1,
return it.
```

```
    2. Pick Pivot:
       Choose one number from the list.
Let's say we pick the first one.
       Call this 'p'.

    3. Partition (The Sorting Step):
       Create a list 'Left' for numbers
smaller than p.
       Create a list 'Right' for numbers
bigger than p.

    4. Recurse:
       SortedLeft = QuickSort(Left)
       SortedRight = QuickSort(Right)

    5. Combine:
       Return SortedLeft + [p] +
SortedRight
```

Simple Example

Let's sort: **[3, 1, 4, 2]**

1. **Pick Pivot:** Let's pick **3**.
2. **Partition:**
   - Is 1 smaller than 3? Yes. Put in **Left**.

- Is 4 smaller than 3? No. Put in **Right**.
- Is 2 smaller than 3? Yes. Put in **Left**.
- **Left List:** [1, 2]
- **Right List:** [4]

3. **Recurse:**
   - Sort [1, 2] (Pick 1 as pivot $\rightarrow$ [1, 2] is sorted).
   - Sort [4] (It's just one number, so it's sorted).
4. **Combine:**
   - [1, 2] + [3] + [4] $\rightarrow$ **[1, 2, 3, 4]**.

Part 2: Parallel Quick Sort (From Figure 12)

Now let's look at the code in **Figure 12**. The logic is the same, but the notation is fancier because it is precise about doing things in parallel.

The Symbols in Figure 12

- S: The Set (or sequence) of numbers we want to sort.
- $|S|$: The size of the set.

- λ **(Lambda)**: This is a fancy way of writing a "rule" or a small function.
    - *Example:* $\lambda x.\,(x < p)$ translates to: "Check a number x. Is x smaller than p? If yes, keep it."
- **FILTER**: This creates a new list by keeping only the items that follow the rule.
- $||$: Run the left code and right code at the same time.

Parallel Pseudocode Explanation

Here is the translation of Figure 12:

```
QUICKSORT(S):
    if (|S| = 0) then return S;        (If
empty, stop)
    else
        p ← SELECTPIVOT(S);
(Pick a random number 'p')

        e ← FILTER(S, equal to p);
(Find all copies of 'p')
```

```
        l ← QUICKSORT(smaller than p) ||
r ← QUICKSORT(bigger than p)
        (Sort the small numbers AND sort
the big numbers at the same time!)

        return FLATTEN(l, e, r)
(Stick them together: Left-Middle-Right)
```

Part 3: What is the math in Section 5.2?

The text below Figure 12 (Section 5.2) is trying to prove **why** this is fast.

Quick Sort is risky. If you pick a bad Pivot (like the smallest number every time), the sort is very slow. If you pick a good Pivot (the middle number), it is very fast. Since we pick pivots randomly, the text uses probability to prove it will *likely* be fast.

Here are the symbols from the math section explained simply:

- A_{ij}: This is a "variable" that acts like a switch.

    - Think of all the numbers lined up in order. Let i be a small number and j be a big number.
    - $A_{ij} = 1$ if number i and number j ever get compared to each other.
    - $A_{ij} = 0$ if they never get compared.
    - *Why does this matter?* The total time the algorithm takes depends on how many comparisons we make.
- $p(A_{ij} = 1)$: The probability (chance) that these two numbers get compared.
    - The text calculates this as $2/(j - i + 1)$.
    - *Intuition:* If two numbers are far apart in value (large difference between i and j), it is very unlikely they will ever be compared, because a pivot will likely land between them and separate them into different lists first.
- **Chernoff Bounds (Theorem 5.1)**:
    - **What it is:** A statistical rule.

- **What it says here:** It proves that it is extremely unlikely ("exponentially low probability") that the algorithm will run slowly. It guarantees that "Span" (the parallel time) is usually around $O(\log n)$.

Summary of the Parallel Gain:

In sequential Quick Sort, you sort the Left, wait for it to finish, then sort the Right.

In Figure 12, the $||$ symbol means the computer creates a new process. One processor handles the small numbers, and another handles the big numbers **simultaneously**.