# Exam Assignments V04

chengchengguo 183090

## 1. Explain how **divide and conquer** algorithms can be parallelized with **tasks** in OpenMP.

divide and conquer strategy can be utilized in recursive tasks.
how merge sort uses divide-and-conquer:[1]

- Divide
  divide the array in to 2, 4, 8 smaller arrays, in a tree hierarchie
- Conquer by recursively sorting the subarrays in each of the two subproblems created by the divide step. That is, recursively sort the subarray and recursively sort the subarray .array[p..q]array[q+1..r]
- Combine by merging the two sorted subarrays back into the single sorted subarray .array[p..r]

OpenMP can be used after the array divided into 2, these 2 array can run parallel.

```
    const int size_b = n - size_a;
#pragma omp task if (size_a > 10000) // make task if size_a
    merge_sort(arr, size_a); // parallel recursive call
    merge_sort(arr + size_a, size_b); // recursive call
```

## 2. Describe some **ways** to **speed up merge sort.**

1) for small array we use this insertion_sort, in this case merge_sort would be expensive

```
void merge_sort(int *arr, int n) {
  if (n > 1) {
    if (n < 32) { // insertion sort for n smaller than 32
      insertion_sort(arr, n);  // efficient for small array
      return;
    }
    const int size_a = n / 2;
```

2) this no wait can also save time but not that much

[1] Merge sort algorithm overview (article) | Khan Academy

```
}

void merge_sort_run(int *arr, int n) {
#pragma omp parallel
#pragma omp single nowait
  merge_sort(arr, n);
}
```

3) use stack to save array

```
if (n < 8192) { // allocate array on the stack for small n
  // sometimes called _alloca() or _malloca(), or "int c[n]"
  int* c = (int*) alloca(n * sizeof(int));
  merge(arr, arr + size_a, c, size_a, size_b, n);
  memcpy(arr, c, sizeof(int) * n); // copying can be tuned away.
  return;
```

4) parallize the merge sort

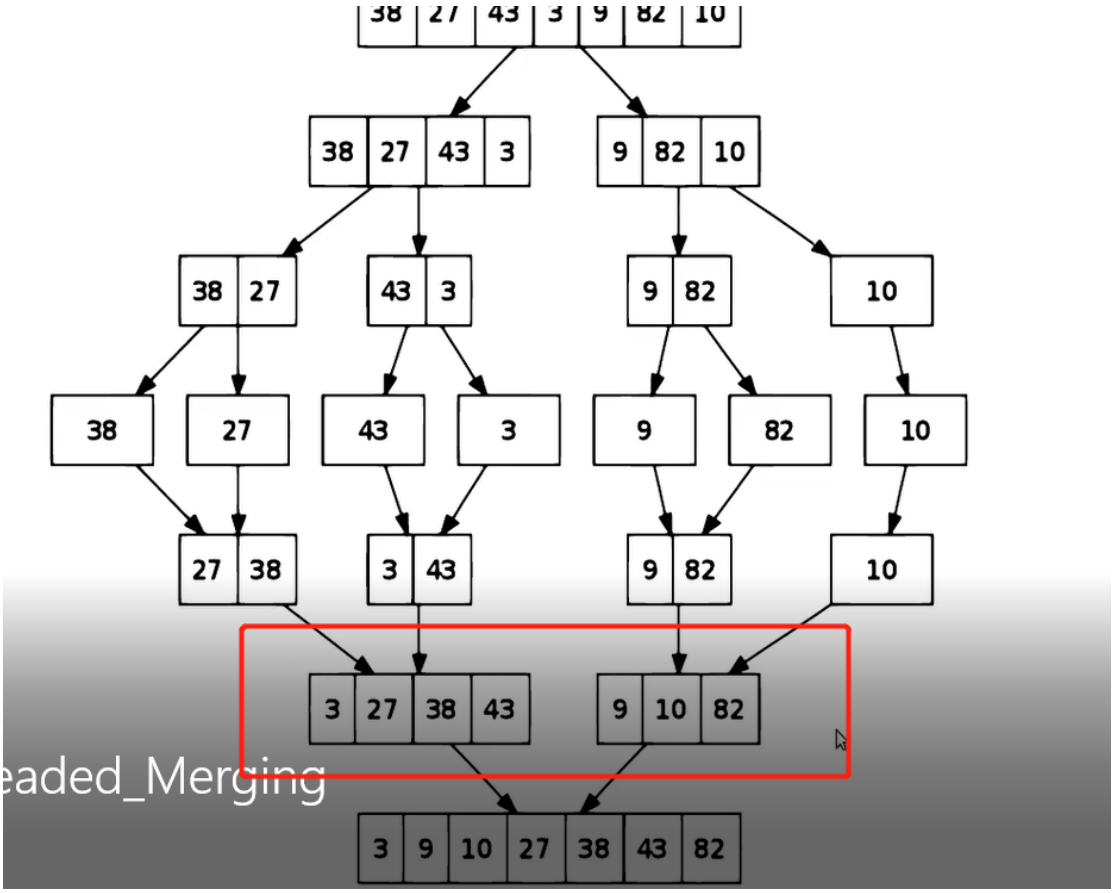
```
void merge_sort_run(int *arr, int n) {
#pragma omp parallel
  {
#pragma omp single nowait
    {
      merge_sort(arr, n);
    }

  }
}
```

5)

```
    const int size_b = n - size_a;
#pragma omp task if (size_a > 10000) // make task if size_a > 10000, one
    merge_sort(arr, size_a); // parallel recursive call
    merge_sort(arr + size_a, size_b); // recursive call
#pragma omp taskwait // wait until both subarrays are sorted
```

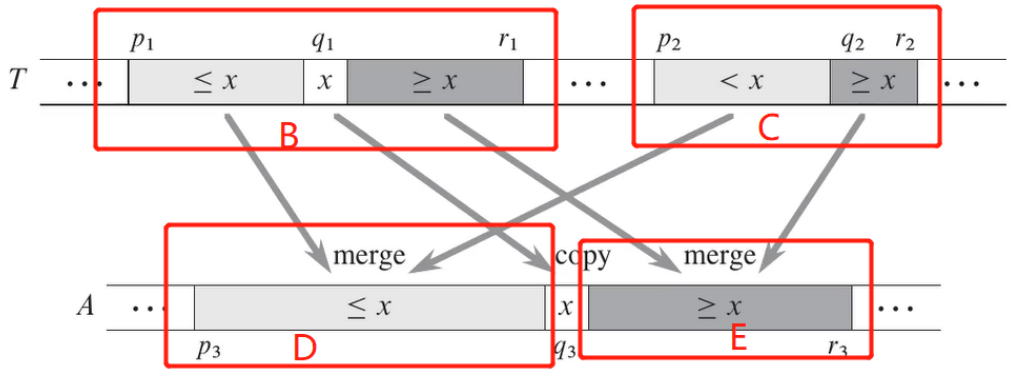
## 3. What is the idea behinbd **multithreaded merging?**

in worst case this merge step can only run in 1 thread




now we want the merge phase parallel run

that means in this step




we want to merge B and C, we find a value x in the middle of B(longer array), and find the part in C that < x, then we can merge the value that < x in B and C together, this is not related with E( the parts that >x)

the > x and <x part can be done in different threads.

4. Read What every systems programmer should know about concurrency.

[What every systems programmer should know about concurrency (bitbashing.io)]

## Discuss **two things** you find particularly interesting.

. Creating some sense of order between threads is a team effort of the hardware, the compiler, the programming language, and your application.

1)

I find this part " 12. Cache effects and false sharing" is interesting.

I had already know false sharing happens when 2 core read/ write a same value. And we can use locks like mutex to make sure only one core modify this value at one time. But in this article, it mentioned :

- *If multiple readers—each running on a different core—simultaneously take the lock, its cache line will "ping-pong" between those cores' caches.*
- *when it occurs between unrelated variables that happen to be placed on the same cache line*

and to avoid it, the article says we can pad atomic variables with a cache line of unshared data, but this is obviously a large space-time tradeoff.

2) another part I find interesting is the method "atomic"

- *"Something is atomic if it cannot be divided into smaller parts."*

We're usually using atomic operations to avoid locks in the first place. Atomic operations are sequences of instructions that guarantee atomic accesses and updates of shared single word variables. This means that atomic operations cannot protect accesses to complex data structures in the way that locks can, but they provide a very efficient way of serializing access to a single word.[2]

[2] Atomic Operations - IBM Documentation