Parallel Programming

Odd - Even Merge Sort

Marcos Alonso and Carlos Vega de Alba

08 / 05 / 2024

Introduction

In this project, we delve into the Odd-Even Merge Sort algorithm and its parallelization using OpenMP. While Merge Sort stands as one of the cornerstone sorting algorithms due to its efficiency and simplicity, the Odd-Even Merge Sort remains relatively obscure despite its notable advantages.

The beauty of Odd-Even Merge Sort lies in its unique approach to sorting, where it operates by iteratively merging adjacent sorted sequences in a parallel manner. Unlike the traditional Merge Sort, which relies on sequential merging of subarrays, the Odd-Even Merge Sort inherently lends itself to parallelization due to its structured merging pattern. This characteristic makes it particularly appealing for modern parallel computing architectures, where exploiting concurrency is essential for achieving optimal performance.

We have chosen to explore the Odd-Even Merge Sort not only for its inherent efficiency but also for its educational value; while many are familiar with Merge Sort, fewer are acquainted with its odd-even variant and its potential benefits. In this paper, we present our implementation and analysis of the parallel Odd-Even Merge Sort algorithm using OpenMP, aiming to elucidate its parallel structure, performance characteristics, and scalability.

SEQUENTIAL ALGORITHM

We are now going to explain the sequential algorithm in C that we implemented to get the Odd-Even Merge Sort.

```c
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Use: %s <number of elements>\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);
    if (n <= 0) {
        fprintf(stderr, "Invalid number. Use a positive number.\n");
        return 1;
    }

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        fprintf(stderr, "Malloc failed.\n");
        return 1;
    }

    srand(time(NULL));
    int i;
    for (i = 0; i < n; i++) {
        arr[i] = rand() % INT_MAX;
    }

    int * newArr;
    int newSize;
    newArr = sizeFix(arr, n, &newSize);

    double start_time = omp_get_wtime();
    sort(arr, newArr, n, newSize);
    double end_time = omp_get_wtime();


    if (isSorted(newArr, n) == 1)
    {
        printf("Total time: %f \n", end_time - start_time);
        if (n <= 100) {
        printf("\n");
```

```
        }
    }

    free(newArr);
    free(arr);
    return 0;
}
```

We parse and check the input, and we generate random numbers between 0 and INT_MAX. Then we call the sort method. We measure the time using omp_get_wtime() right before and after the execution of the sorting algorithm.

Also, if the array is not a power of two, we adjust it to be a power of two. We fill it with INT_MAX, as it won't affect the final order of the array.

```
void oddEvenMergeSort(int *arr, int n, int level) {
    if (n > 1) {
        int m = n / 2;

        oddEvenMergeSort(arr, m, level);

        oddEvenMergeSort(arr + m, n - m, level);

        oddEvenMerge(arr, n);
    }
}

void sort(int *arr, int *newArr, int n, int newSize) {
    int i;
    oddEvenMergeSort(newArr, newSize, 0);

    // Original elements
    for (i = 0; i < n; i++) {
        arr[i] = newArr[i];
    }
}
```

In sort we call the main sort method, oddEvenMergeSort, and we copy the original elements.

In oddEvenMergeSort, we make the recursion by calling it in both halves of the array, and then merging. The base case to stop the recursion is when the array is only one element.

```
void oddEvenMerge(int *arr, int n) {
   if (n == 1) return;
   if (n == 2) {
       compare(arr, 0, 1);
       return;
   }
   else {
       int *odd = (int *)malloc(sizeof(int) * (n / 2));
       int *even = (int *)malloc(sizeof(int) * (n / 2));
       int i;
       oddEvenSplit(arr, odd, even, n);

       oddEvenMerge(odd, n / 2);

       oddEvenMerge(even, n / 2);

       oddEvenJoin(arr, odd, even, n);

       for (i = 1; i < n / 2; i++) {
           compare(arr, 2 * i - 1, 2 * i);
       }

       free(odd);
       free(even);
   }
}
```

To merge the array, we use the odd even split. We then call the same method in the odd array and in the even array. We stop the recursion when n is one, or when it is two. In this case we compare both elements. At the end, we make the loop to compare the elements that are still unordered. We free the memory allocated.

This implementation follows a depth-first strategy, and for every recursive merge you are doing memory allocation and de-allocation. The algorithm could also have been implemented using a BFS-strategy. Then it would have been possible to only use two arrays. (at most)

```c
int isSorted(int *arr, int n) {
    int i;
    for (i = 0; i < n - 1; i++) {
        if (arr[i] > arr[i + 1]) {
            return 0;
        }
    }
    return 1;
}

void compare(int *arr, int a, int b) {
    if (arr[a] > arr[b]) {
        int tmp = arr[a];
        arr[a] = arr[b];
        arr[b] = tmp;
    }
}

void oddEvenSplit(int *arr, int *odd, int *even, int n) {
    int i;
    for (i = 0; i < n/2; i++) {
        odd[i] = arr[2 * i + 1];
        even[i] = arr[2 * i];
    }
}

void oddEvenJoin(int *arr, int *odd, int *even, int n) {
    int i;
    for (i = 0; i < n/2; i++) {
        arr[2 * i + 1] = odd[i];
        arr[2 * i] = even[i];
    }
}
```

We have this auxiliar methods, one to split into odd and even and one to re-join. Compare method compare two values and swap them if it’s the case. isSorted is only to check if the final array is sorted.

Parallel Algorithm

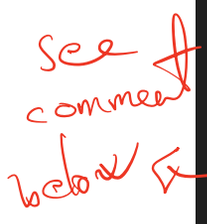
We now are in the interesting part. This algorithm is thought to be parallelized. Each halve of the array is sorted in an independent way, so when we divide the array in halves, we create a task for each one. We do this again and again in a recursive way, and we stop when the level of depth reaches the limit we imposed.

```
int main(int argc, char *argv[]) {
   if (argc != 4) {
       fprintf(stderr, "use: %s <número de elementos> <limit> <threads>\n",
argv[0]);
       return 1;
   }

   int n = atoi(argv[1]);
   int limit = atoi(argv[2]);
   int nthreads = atoi(argv[3]);
   omp_set_num_threads(nthreads);
   if (n <= 0) {
       fprintf(stderr, "Invalid number. Write a positive number\n");
       return 1;
   }

   int *arr = (int *)malloc(n * sizeof(int));
   if (arr == NULL) {
       fprintf(stderr, "Malloc failed.\n");
       return 1;
   }

   srand(time(NULL));
   int i;
   for (i = 0; i < n; i++) {
       arr[i] = rand() % INT_MAX;
   }

   int * newArr, newSize;
   newArr = sizeFix(arr, n, &newSize);

   double start_time = omp_get_wtime();
   sort(arr, newArr, n, limit, newSize);
   double end_time = omp_get_wtime();

   if (isSorted(newArr, n) == 1)
   {
```

```
        printf("Total time: %f \n", end_time - start_time);
        if (n <= 100) {
        printf("\n");
        }
    }

    free(newArr);
    free(arr);
    return 0;
}
```

The main is very similar, but we add two parameters: limit depth of the recursion to stop using parallel approach, and number of threads.

```
void oddEvenMergeSort(int *arr, int n, int *level, int limit) {
    if (n > 1) {
        int m = n / 2;
        int localLevel = *level;

        if (localLevel < limit)
        {
            localLevel += 1;
            #pragma omp task shared(arr)
            oddEvenMergeSort(arr, m, &localLevel, limit);

            #pragma omp task shared(arr)
            oddEvenMergeSort(arr + m, n - m, &localLevel, limit);

            #pragma omp taskwait
            oddEvenMerge(arr, n, localLevel, limit);
        }
        else {
            oddEvenMergeSort(arr, m, &localLevel, limit);

            oddEvenMergeSort(arr + m, n - m, &localLevel, limit);

            oddEvenMerge(arr, n, 0, localLevel);
        }
    }
}

void sort(int *arr, int *newArr, int n, int limit, int newSize) {

    #pragma omp parallel
    {
        int i;
```



```
        int *level = (int *)malloc(sizeof(int));
        *level = 0;
        #pragma omp single
        oddEvenMergeSort(newArr, newSize, level, limit);

        // Original elements
        #pragma omp for schedule(static)
        for ( i = 0; i < n; i++) {
            arr[i] = newArr[i];
        }

    }

}
```

Then, we perform the algorithm, but using omp tasks. To make this, a **single** thread creates the tasks, and then one thread take the task from the queue and work on it. We put a depth limit, because it is not computationally optimum when numbers are very big to make a big parallel recursion tree. So level is increased each recursion, and when it reach the limit, it continues in a sequential approach. Why continue with the odd-even strategy? You could just have run the standard merge sort.

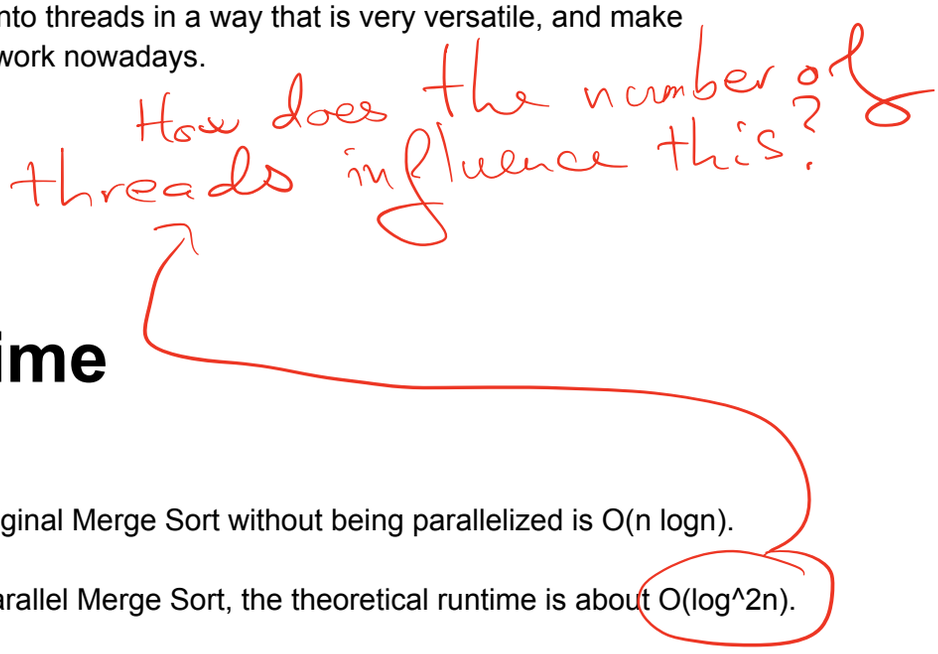
```
void oddEvenMerge(int *arr, int n, int level, int limit) {
    if (n == 1) return;
    if (n == 2) {
        compare(arr, 0, 1);
        return;
    }
    else {
        int *odd = (int *)malloc(sizeof(int) * (n / 2));
        int *even = (int *)malloc(sizeof(int) * (n / 2));
        int i;
        oddEvenSplit(arr, odd, even, n);

        oddEvenMerge(odd, n / 2, level, limit);

        oddEvenMerge(even, n / 2, level, limit);

        oddEvenJoin(arr, odd, even, n);

        for (i = 1; i < n / 2; i++) {
            compare(arr, 2 * i - 1, 2 * i);
        }

        free(odd);
        free(even);
```

```
    }
}
```

Here, we tried to parallelize the loop of comparisons, but it was not complex enough to be worth it. The split and the join are as well too simple to parallelize them. So, it is the same approach as the sequential one.

The auxiliar methods are the same than in the sequential approach.

Tasks were very important for this exercise. OpenMP was designed a long time ago, and it was thinked to increase performance of other types of programs. So it was mainly created for array parallelization. But in modern times, other type of programs and algorithms are used. For this, openMP spend more or less 8 years to implement tasks in a huge update. This allow us to share the work into threads in a way that is very versatile, and make OpenMP a reference in parallel work nowadays.



Theory Runtime

The theoretical runtime of the original Merge Sort without being parallelized is O(n logn).

When we use the Odd - Even Parallel Merge Sort, the theoretical runtime is about O(log^2n).

Anyway, as the array increases its size, the memory limitations of being so big make this runtime impossible in our program. This will be caused by the amount of failed reading in memory because as the array gets bigger it is impossible to hold all the possible numbers at the same time in memory. In the next part, the tests and the results, we can observe that when the array is bigger, we get worse performance.

Performance of Parallel vs Sequential algorithms

We will try for values of n to the power of 13 until n to the power of 22 and see how the runtime of each algorithm is.

- **Sequential:**
  First we started with the sequential.

n	Time Sequential
8192	0.012309
16384	0.027045
32768	0.059644
65536	0.129805
131072	0.281754
262144	0.566614
524288	0.695526
1048576	1.250750
2097152	2.675398
4194304	5.848610

Standard mergesort uses ~0.84 sec on the same problem and can sort about 40M elements in 5.6 sec. This should be an indication that your implementation could have been better.

- **Parallel:** Limit = 5

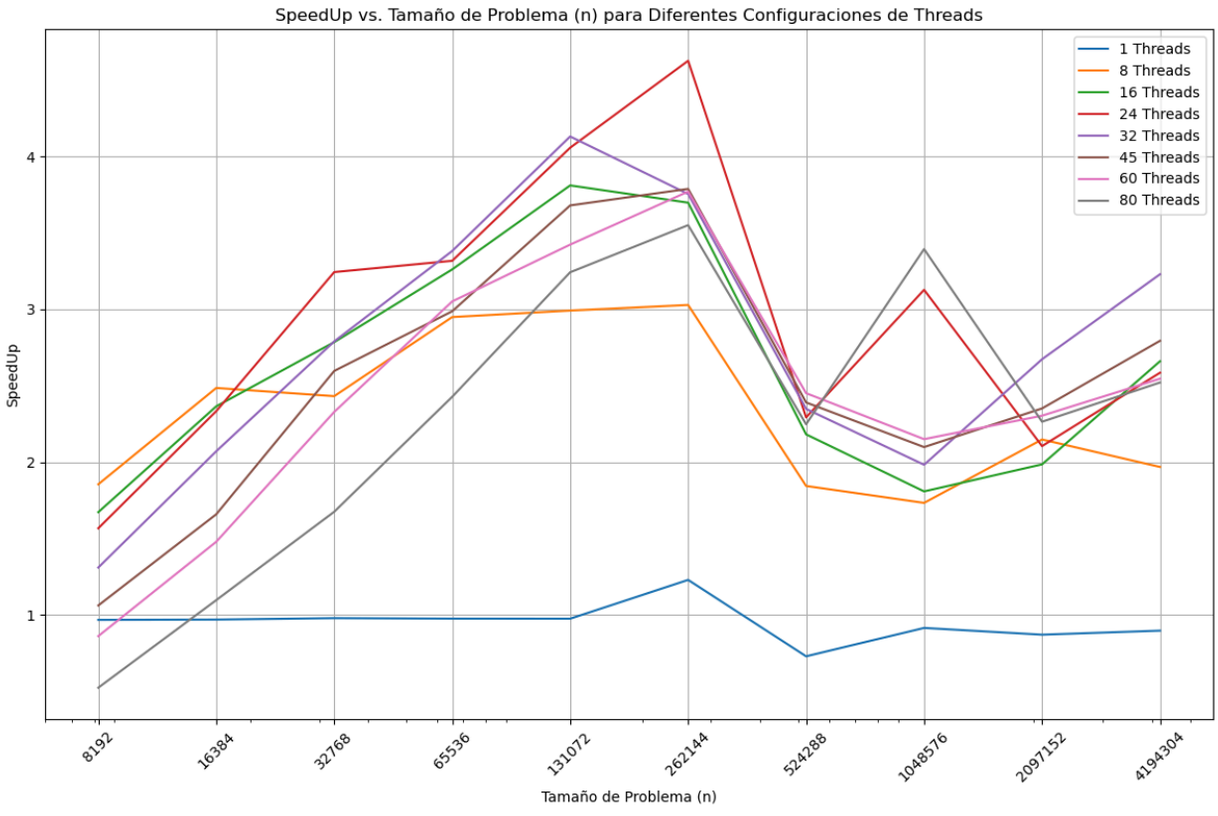
RUNTIMES

n	1 Thread	8 Threads	16 Threads	24 Threads	32 Threads	45 Threads	60 Threads	80 Threads
8192	0.012715	0.006635	0.007358	0.007853	0.009392	0.011584	0.014279	0.023455
16384	0.027881	0.010882	0.011435	0.011590	0.013057	0.016302	0.018282	0.024643
32768	0.060904	0.024527	0.021417	0.018390	0.021380	0.022970	0.025615	0.035597
65536	0.132973	0.044018	0.039809	0.039134	0.038392	0.043461	0.042543	0.053475
131072	0.288749	0.094190	0.073951	0.069454	0.068219	0.076582	0.082313	0.086906
262144	0.460772	0.187122	0.153260	0.122503	0.150938	0.149608	0.150319	0.159599
524288	0.952784	0.377141	0.318810	0.303368	0.296188	0.290969	0.283742	0.309422
1048576	1.366191	0.721576	0.691613	0.399966	0.630833	0.596058	0.581661	0.368572
2097152	3.070086	1.245434	1.347650	1.271064	1.000874	1.137857	1.161265	1.181467
4194304	6.514826	2.971493	2.198677	2.262523	1.811260	2.093566	2.297263	2.320062

SPEEDUP

n	1 Thread	8 Threads	16 Threads	24 Threads	32 Threads	45 Threads	60 Threads	80 Threads
8192	0.968069	1.855162	1.672873	1.567426	1.310583	1.062586	0.862035	0.524792
16384	0.970015	2.485297	2.365107	2.333477	2.071303	1.658999	1.479324	1.097472
32768	0.979312	2.431769	2.784891	3.243284	2.789710	2.596604	2.328479	1.675534
65536	0.976176	2.948907	3.260695	3.316937	3.381043	2.986701	3.051148	2.427396
131072	0.975775	2.991337	3.810009	4.056699	4.130140	3.679115	3.422959	3.242055
262144	1.229706	3.028046	3.697077	4.625307	3.753952	3.787324	3.769410	3.550235
524288	0.729993	1.844207	2.181632	2.292681	2.348259	2.390378	2.451262	2.247823
1048576	0.915502	1.733359	1.808454	3.127141	1.982696	2.098370	2.150307	3.393502
2097152	0.871441	2.148165	1.985232	2.104849	2.673062	2.351260	2.303865	2.264471
4194304	0.897738	1.968240	2.660059	2.584995	3.229028	2.793611	2.545904	2.520885

Plot SpeedUp:



The best speedUp for this case is around 5 using 24 threads for an array size of 2^18.

- **Parallel:** Limit = 7

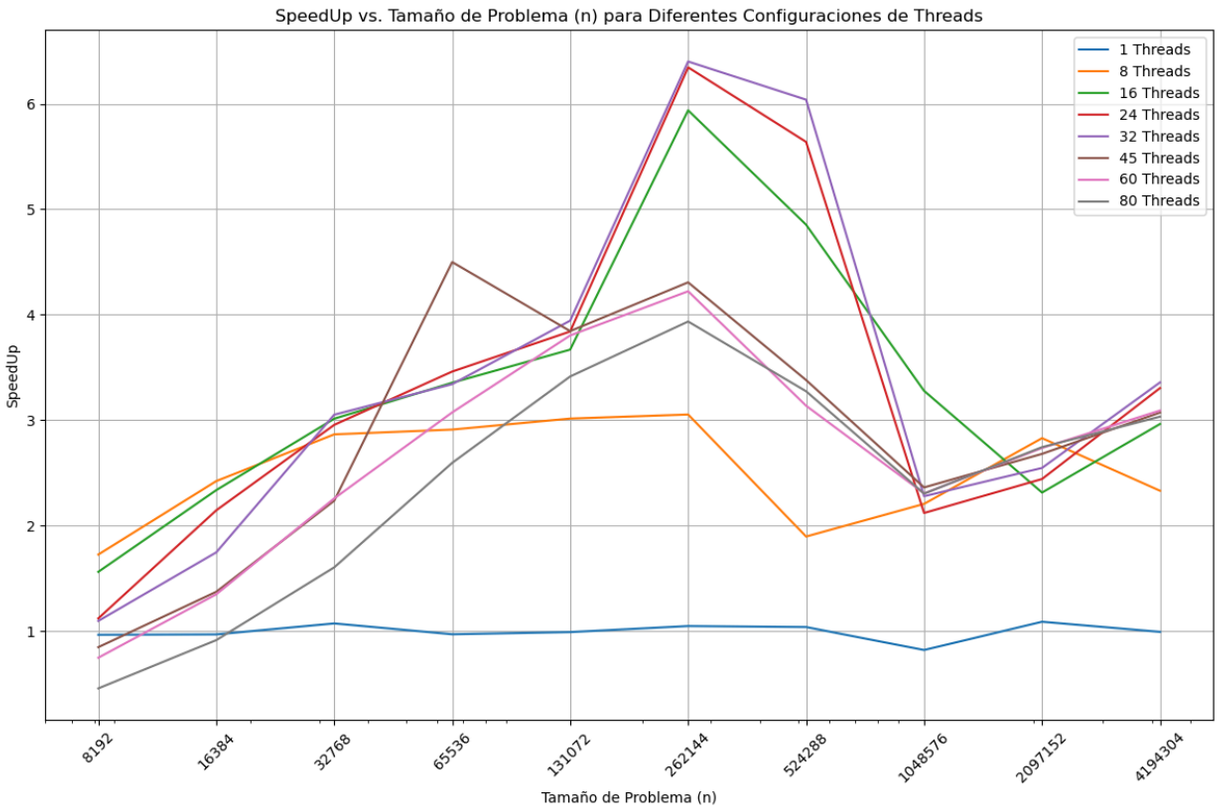
RUNTIME:

n n	**1 Thread**	**8 Threads**	**16 Threads**	**24 Threads**	**32 Threads**	**45 Threads**	**60 Threads**	**80 Threads**
8192	0.012743	0.007134	0.007883	0.010984	0.011223	0.014510	0.016448	0.026890
16384	0.027902	0.011169	0.011578	0.012612	0.015488	0.019702	0.020048	0.029550
32768	0.057034	0.021368	0.020316	0.020718	0.020063	0.027376	0.027112	0.038147
65536	0.133683	0.044604	0.038695	0.037511	0.038854	0.028860	0.042239	0.050018
131072	0.284453	0.093561	0.076892	0.073453	0.071537	0.073384	0.074174	0.082624
262144	0.580997	0.199548	0.102645	0.096050	0.095221	0.141503	0.144340	0.154850
524288	0.877421	0.480748	0.187971	0.161806	0.151073	0.269854	0.290774	0.278600
1048576	1.565948	0.583247	0.392639	0.606942	0.564639	0.544757	0.558111	0.558603
2097152	2.814743	1.083948	1.325228	1.255672	1.203929	1.144128	1.120934	1.117606
4194304	6.130575	2.613237	2.055660	1.843741	1.814364	1.983121	1.970795	2.008480

SPEEDUP:

n n	1 Thread	8 Threads	16 Threads	24 Threads	32 Threads	45 Threads	60 Threads	80 Threads
8192	0.966805	1.726941	1.562857	1.121631	1.097746	0.849070	0.749027	0.458163
16384	0.970074	2.423404	2.337796	2.146131	1.747611	1.373820	1.350110	0.915973
32768	1.073851	2.866249	3.014668	2.956173	3.052684	2.237215	2.258999	1.605526
65536	0.971380	2.911331	3.355912	3.461838	3.342178	4.499550	3.074339	2.596205
131072	0.991935	3.015776	3.669549	3.841354	3.944239	3.844966	3.804015	3.414976
262144	1.049045	3.054363	5.937864	6.345570	6.400815	4.307273	4.222613	3.936015
524288	1.039788	1.897734	4.853578	5.638431	6.039014	3.380836	3.137598	3.274702
1048576	0.822148	2.207370	3.278946	2.121194	2.280115	2.363333	2.306785	2.304753
2097152	1.089883	2.830154	2.314877	2.443106	2.548107	2.681291	2.736771	2.744921
4194304	0.994024	2.331951	2.964469	3.305204	3.358720	3.072904	3.092123	3.034106

Plot:



The best speedUp for this case is almost 7 using 32 threads for an array size of 2^18.

- **Parallel:** Limit = 10

RUNTIMES

n n	**1 Thread**	**8 Threads**	**16 Threads**	**24 Threads**	**32 Threads**	**45 Threads**	**60 Threads**	**80 Threads**
8192	0.013419	0.022695	0.034035	0.040224	0.038754	0.049152	0.048832	0.057095
16384	0.028519	0.019709	0.035749	0.042201	0.049314	0.052145	0.051905	0.064986
32768	0.061594	0.028811	0.035050	0.048630	0.053390	0.059801	0.058181	0.070841
65536	0.133743	0.046510	0.042154	0.048981	0.061564	0.069188	0.072336	0.082963
131072	0.269108	0.099008	0.077528	0.075885	0.084696	0.093406	0.088810	0.115433
262144	0.549051	0.171807	0.156024	0.149488	0.145760	0.131937	0.152772	0.170131
524288	0.901793	0.419489	0.329795	0.297963	0.292748	0.281843	0.257975	0.200283
1048576	1.628336	0.720028	0.653798	0.627067	0.578246	0.559682	0.554138	0.549545
2097152	3.493396	1.332179	1.356280	1.213381	1.185462	1.149629	0.910165	1.102591
4194304	6.303794	2.354831	2.700326	2.302648	2.329231	1.741322	2.104854	2.111439

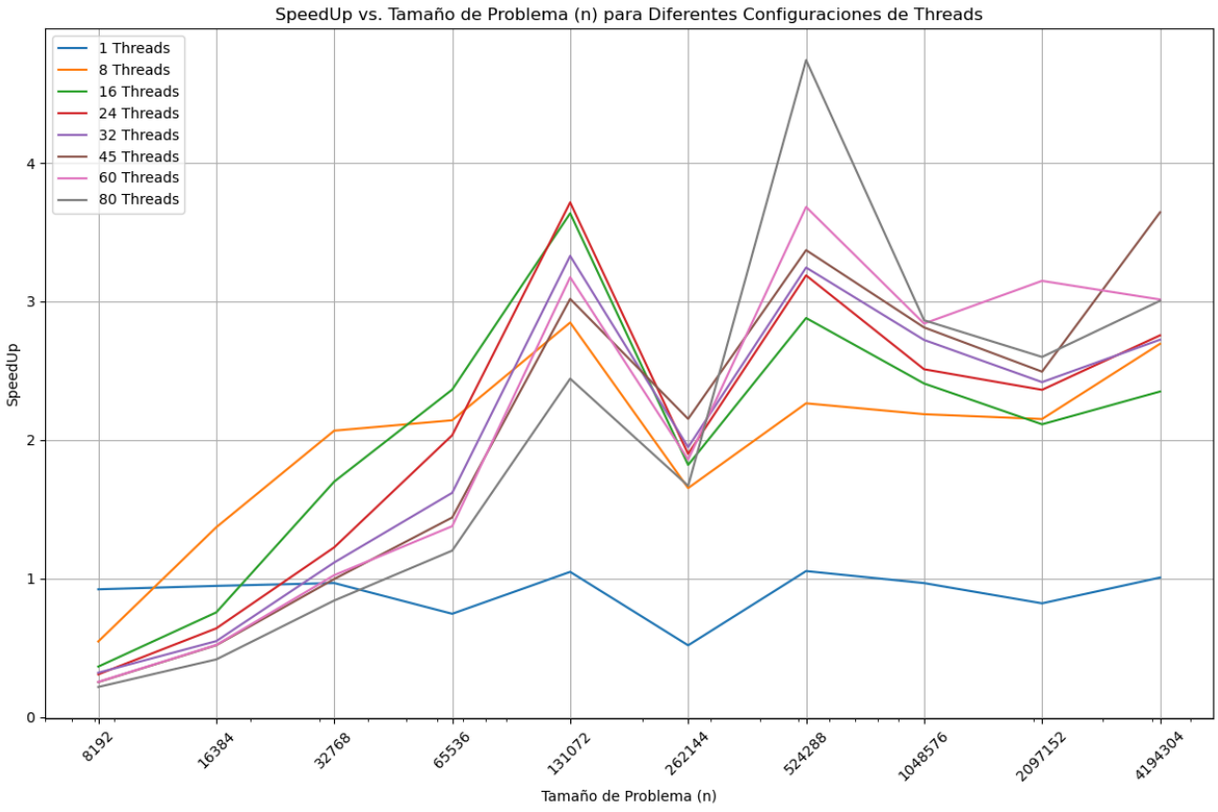
SPEEDUP:

n n	1 Thread	8 Threads	16 Threads	24 Threads	32 Threads	45 Threads	60 Threads	80 Threads
8192	0.921678	0.544966	0.363391	0.307478	0.319141	0.251628	0.253277	0.216621
16384	0.946387	1.369425	0.754986	0.639558	0.547309	0.517595	0.519988	0.415320
32768	0.966815	2.066919	1.699001	1.224553	1.115377	0.995803	1.023530	0.840615
65536	0.744846	2.141862	2.363192	2.033809	1.618121	1.439816	1.377157	1.200752
131072	1.047546	2.847275	3.636144	3.714871	3.328410	3.018040	3.174226	2.442135
262144	0.517045	1.652342	1.819489	1.899042	1.947613	2.151663	1.858220	1.668620
524288	1.053228	2.264169	2.879953	3.187624	3.244408	3.369940	3.681729	4.742260
1048576	0.966528	2.185794	2.407216	2.509832	2.721736	2.812013	2.840146	2.863884
2097152	0.820252	2.150960	2.112738	2.361553	2.417171	2.492512	3.148291	2.598846
4194304	1.006301	2.693831	2.349167	2.754879	2.723438	3.642931	3.013756	3.004357

It is nice that you tried a somewhat difficult problem, but I think you could have achieved more. First, your sequential code is not good, this is evident by how slow it is compared to the standard merge sort alg. The main problem is most likely that you are using recursion instead of a bottom-up approach. If you had been working on the whole array in each step I think it would have improved your speedup. Also, it does not make sense to use a suboptimal sequential algorithm once you decide not to run in parallel. Your analysis is also missing.
All this being said, you still got a speedup that was not too bad, but only compared to the algorithm itself, not compared to sequential merge sort.

20/30

Plot:



The best speedUp for this case is almost 5 using 80 threads for an array size of 2^19.

Conclusion

With this parallel algorithm, we were able to achieve approximately x7 speedup with 32 threads and a limit of 7. This is a huge improvement in comparison with a normal Merge Sort algorithm, which every one knows. This make odd-even merge sort algorithm something special that need to be more present in nowadays programs and education.

OpenMP designed tasks for problems like this, and it is very useful to use them.