Golden Collision Project
Meet-in-the-middle using MPI

Yalda Eftekhari - 21418615
L.G. Paxton - 21419023

course: Parallel Programming

Indhold

1 Project Description **2**

2 Project Layout **2**

3 Input Output Behaviour **5**

4 Results **6**

5 Design Choices **6**

6 Conclusion and possible Improvements **7**

7 Appendix **7**

1 Project Description

Parallel Meet-in-the-Middle Attack for Hash Function Collisions

The goal of this project is to implement a meet-in-the-middle attack algorithm to find collisions for two given hash functions. A *collision* occurs when two different inputs from a set of numbers to two hash functions produce the same output. Formally, let:

$$f, g : \{0,1\}^n \to \{0,1\}^n, \tag{1}$$

such that we find $x, y \in \{0,1\}^n$ satisfying:

$$f(x) = g(y). \tag{2}$$

A *golden collision* is defined as a collision where, for a given function $\pi : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, the condition

$$\pi(x, y) = 1 \tag{3}$$

is satisfied.

Parallel Meet-in-the-Middle Attack

The parallel meet-in-the-middle attack is designed to efficiently accelerate the process of finding collisions by dividing the search space into two main components: x-values and y-values.

The x-values are split into multiple smaller sections, and the intermediate results are stored in dictionaries. These dictionaries are distributed across different processing nodes, allowing parallel computation. Each node handles a subset of the x-values and prepares the corresponding entries in its dictionary.

On the other side, y-values are processed by the processors once they are available for a job. Each processor compares the $g(y)$, with the y-values, against the entries in the distributed dictionaries of x-values and their associated key, $f(x)$ to identify potential collisions. To maximize efficiency, the y-values are sent to the dictionaries on different processing nodes in a strategic manner, minimizing communication overhead while ensuring thorough collision checks.

As multiple collisions are identified through this parallel approach, each collision is checked to see if it satisfies the golden rule, i.e., whether $\pi(x, y) = 1$.

Project Focus

This project focuses on implementing the parallelization of the meet-in-the-middle attack algorithm. The main objectives are to optimize memory usage on each process and improve the overall computational speed.

2 Project Layout

The total search space $N = 2^n$ is divided among `mpi_size` processes. The goal is to divide some data among multiple processes as evenly as possible while accounting for any remainder when the data size is not perfectly divisible by the number of processes.
The first term, mpi_rank * subdict_size, calculates the starting index assuming all processes get an equal chunk size (subdict_size). The second term, (mpi_rank >= rest ? rest : mpi_rank), adjusts for the remainder (rest): If mpi_rank < rest, this process gets one extra element from the remainder. If mpi_rank >= rest, this process does not get an extra element. So, ach process is responsible for a specific range of values, determined by:

```
u64 proc_start_val = mpi_rank * subdict_size + (mpi_rank >= rest ? rest :
    mpi_rank);
u64 proc_end_val = proc_start_val + subdict_size + (mpi_rank < rest ? 1 :
    0);
```

Below code calculates the size of data each process will handle, but doubling the base size unnecessarily inflates the allocation. This leads to increased memory usage and communication overhead, making it a bottleneck in distributed processing. Dynamically allocating based on actual needs can optimize resource utilization and improve performance. We didn't have enough time to implement this with alltoallv so we are using fixed send and receive buffer sizes.

```
u64 dict_size_for_each_process = (u64)((N / mpi_size)) * 2;   // example
    10
```

Info array is defined to give its first element is size of the send and receive buffer so all processes know it and this ensures that all processes will have the same size and partitioning for MPI_Alltoall. In the rank==zero process, the first value of the info array is initialized. For each subsequent index i, the starting position of the buffer belonging to that process is determined. Since all buffers have a size equal to $info\_array[0]$, the starting position for process i is multiplication of I and $info\_array[0]$.

```
u64 * info_array = malloc(sizeof(u64)*(mpi_size + 1));
if (mpi_rank == 0){
    info_array[0] = dict_size_for_each_process;
    for (int i = 0; i < mpi_size; i++){
        info_array[i+1] = i * info_array[0];
    }
}
```

We use broadcasting to send these data to all processes. and the send buffer will be initialized with size of $info\_array[0]$.

```
MPI_Bcast(info_array, mpi_size + 1, MPI_UNSIGNED_LONG_LONG, 0,
    MPI_COMM_WORLD);
struct entry *send_buffer = calloc(info_array[0] * mpi_size ,sizeof(struct
    entry));
```

This part of the code parallelizes the computation of z over the range proc_start_val, proc_end_val using OpenMP's #pragma omp parallel for directive, which distributes iterations of the loop across multiple threads to utilize all available CPU cores. For each value of x in the range, the function f(x) is computed to produce z. The value of z determines the batch or process it belongs to, calculated as batch_id=z%mpi_size. The results (x,z) are stored in a shared buffer (send_buffer) at positions indexed by info_array[batch_id+1] which is the index of first empty space in the send buffer belonging to the process batch_id. To ensure thread safety during updates to shared data structures (send_buffer and info_array), critical sections are used with #pragma omp critical. This prevents race conditions by allowing only one thread at a time to execute the enclosed block, ensuring correctness while maintaining parallelism for most of the loop.

```
#pragma omp parallel for
for (u64 x = proc_start_val; x < proc_end_val; x++) {
    u64 z = f(x);
    u64 batch_id = z % mpi_size;
    #pragma omp critical
    {
        send_buffer[info_array[batch_id+1]].v = x;
        send_buffer[info_array[batch_id+1]].k = z;
```

```
            info_array[batch_id+1]++;
        }
    }
```

receive buffer has also same size as send buffer. here we use MPI_Alltoall to perform an all-to-all communication where each process sends its send_buffer (of size dict_size_for_each_process * sizeof(struct entry)) to every other process and receives data into rec_buffer, which is pre-allocated to hold data from all processes (info_array * mpi_size). After receiving, the code iterates through rec_buffer, checking for valid entries (non-zero keys or values, we are considering this condition because we are using mpi_alltoall and we have many empty spaces in our buffer initialized by zero so we should ignore them and the probability of having key and value equal to 0 is low so we just ignore it.), and inserts them into a dictionary using dict_insert. Finally, it frees the memory allocated for both the send and receive buffers. And we add a barrier here to make sure all processes will reach here and none of them will start next part before filling the actual hash table.

```
struct entry *rec_buffer = calloc(info_array[0] * mpi_size
    ,sizeof(struct entry));
MPI_Alltoall(send_buffer,
    dict_size_for_each_process * sizeof(struct entry), MPI_BYTE,
    rec_buffer, dict_size_for_each_process * sizeof(struct entry),
    MPI_BYTE, MPI_COMM_WORLD);

for (int filling = 0; filling < info_array[0] * mpi_size ; filling ++){
    if(rec_buffer[filling].k != 0 || rec_buffer[filling].v != 0)
        dict_insert(rec_buffer[filling].k, rec_buffer[filling].v);
}
free(send_buffer);
free(rec_buffer);
MPI_Barrier(MPI_COMM_WORLD);
```

For second loop of the program we do almost same thing as we did before but here we don't need to save receive buffer anywhere and we can do our comparison with the receive buffer itself.

And finally to find the golden collision, This code uses OpenMP to parallelize a loop that processes entries in rec_buffer2. For each entry, it checks if the key (y) or value (z) is non-zero and probes a dictionary (dict_probe) to find potential matches stored in x. The number of matches (nx) is added to ncandidates using a reduction clause for thread-safe accumulation. If a good pair is found (is_good_pair), it is stored in the result arrays k1 and k2, provided the result count (nres) is below the maximum allowed (maxres). Critical sections ensure thread safety when printing errors, updating results, or incrementing nres. This efficiently processes data in parallel while maintaining correctness.

```
#pragma omp parallel for reduction(+:ncandidates) shared(nres, k1, k2)
for (u64 i = 0; i < info_array[0] * mpi_size; i++) {
    u64 y = rec_buffer2[i].k;
    u64 z = rec_buffer2[i].v;
    if ((y != 0 || z != 0)) {
        int nx = dict_probe(y, 256, x);
        if (nx < 0) {
            #pragma omp critical
            {
                printf("Process %d: Error in dict_probe\n", mpi_rank);
            }
            continue;
        }
```

```
        ncandidates += nx;
        for (int j = 0; j < nx; j++) {
            if (is_good_pair(x[j], z)) {
                #pragma omp critical
                {
                    if (nres < maxres) {
                        k1[nres] = x[j];
                        k2[nres] = z;
                        printf("SOLUTION FOUND!\n");
                        nres += 1;
                    }
                }
            }
        }
    }
}
```

3 Input Output Behaviour

Code Compilation

The following command was used to compile the program:

```
make
make run NP=number of processes n=number to find golden collision
```

The input of the project as a whole is handled by the `int argc` and the `char **argv` variables. An example input is the following:

```
RUN_ARGS_4 = --n 4 --C0 f9bb5ad6381c95db --C1 bfdb212e064b9544
RUN_ARGS_8 =  --n 8 --C0 aede0f853ae3991b --C1 6cfab714ef791bbf
RUN_ARGS_16 = --n 16 --C0 26de6c0be3898390 --C1 84a334d5ac314818
RUN_ARGS_20 = --n 20 --C0 017ec9921b7dae7a --C1 068901c1ef08132b
RUN_ARGS_24 = --n 24 --C0 4c143361d238f251 --C1 81d8694e0ef8fdef
RUN_ARGS_25 = --n 25 --C0 0218ba08bc8a3d62 --C1 08503489026b42bb
RUN_ARGS_26 = --n 26 --C0 8a3c2ea7698e4834 --C1 458179c8b6a25485
RUN_ARGS_27 = --n 27 --C0 49cc4e0401935378 --C1 d1330883152d67a5
```

The values `C0 2e039b8486fc5f44` and `C1 ea1235205fb01059` are values taken from the website

$$https://ppar.tme-crypto.fr/<username>/<n>$$

where *<username>* can be *golden_col* and *<n>* can be *1, 2, ..., 25, 26, ..* . `C0` and `C1` are prompt values for the hash functions and values of x and y.

Some of the outputs can be seen in the appendix section. Notice that with different values of n, one has different sizes in dictionaries and different times to fill the dictionaries and search for a collision.

4 Results

Performance Observations

As the value of n increases, we observe that the runtime of the program also increases, which is consistent with expectations. The size of each individual dictionary grows with n, resulting in longer processing times. Additionally, the increased size of n leads to greater communication overhead as more time is required to transmit data between processes. Furthermore, the number of collisions to be verified also rises with larger values of n, further contributing to the overall computational effort. These results are visible in the appendix.

Some example times can be seen in the following table: (np=1 is for sequential code)

np	n	Dictionary Size	Fill Time (s)	Probe Time (s)
1	4	216B	0.0	0.0
1	16	884.7KB	0.0	1.0
1	25	453.0MB	21	63.4
1	27	1.8GB	84.3	254.3
7	4	48B	0.0	0.0
7	16	224.7KB	0.0	0.0
7	25	115.0MB	2.5	5.1
7	25	115.0MB	2.5	5.1
7	27	460.2MB	9.8	20.7

Noteworthy, is that the sequential code has exponential growth in the dictionary size, whereas the output with more processes stabilize the growth in dictionary size. This allows for n to grow more.

The same result can be seen in the time to fill the dictionaries and probe them. One can therefore notice that computing and probing in one process is a lot more time inefficient than having time spent due to communication overload. Naturally, the amount of processes avaible change this factor.

5 Design Choices

A specific design choice that was made was to use the `MPI_Alltoall` function.

The `MPI_Alltoall` function is a collective communication operation in MPI. It enables all processes in a communicator to exchange data with all other processes. Each process sends a unique block of data to every other process and receives a corresponding block of data from each of them. This data is then stored into the original memory location as the sent data. This allows us to easily distribute data to all processes. It also allows for an easy further use of the data as it is all in consecutive memory. For example, when filling the dictionaries (line 266) or when probing the dictionaries for collisions (line 304), the consecutive structure of the receive buffer is utilized.

A further optimization that was chosen was to make use of `OPENMP`. This could be used for parallelizing, using multiple threads, `for` loops that take place within a process.

The `MPI_Bcast` function was used to ensure that indeed all processes make use of the same values, when using the same values was critical, such as in the implementation of `MPI_Alltoall` as the number of the expended values in the receive buffer is critical for the correct implementation.

Noticeable from the results in the appendix is that the dictionary sizes and the fill time for each process is equal. This is due to the fact that the algorithm is of static load balancing technic. Static load balancing is predictable in nature. The problem at hand is one where the amount of work each process will have to do it is foreseeable, hence static load balancing was used.

The final design choice was to section the values x via their modulo `mpi_size` classes of their keys $f(x)$ and storing the data in the dictionary in the according process. This meant that a key could be traced back to the process, and hence dictionary, where a collision was most likely to occur.

6 Conclusion and possible Improvements

In conclusion, the implementation of MPI_Alltoall in this project has significantly improved the performance and scalability of the meet-in-the-middle attack algorithm. By distributing the workload across multiple processes, we have effectively parallelized the dictionary creation and search operations, allowing for faster processing of larger problem sizes. The use of OpenMP for local parallelization further enhances the efficiency of the code, particularly in the dictionary probing phase.
However, there is room for further optimization. A key improvement would be to implement MPI_Alltoallv instead of MPI_Alltoall. This would allow for variable-sized data exchanges between processes, potentially reducing memory usage and communication overhead. MPI_Alltoallv could enable more efficient load balancing by accommodating uneven distributions of keys across processes. One can see an attempt at the implementation of the MPI_Alltoallv in the appendix.

7 Appendix

Results

input: mpirun -np 7 ./golden$_collision --n4 --C02e039b8486fc5f44 --C1ea1235205fb01059
output:

```
Running with n=4, C0=(86fc5f44, 2e039b84) and C1=(5fb01059, ea123520)
Running with n=4, C0=(86fc5f44, 2e039b84) and C1=(5fb01059, ea123520)
Running with n=4, C0=(86fc5f44, 2e039b84) and C1=(5fb01059, ea123520)
Running with n=4, C0=(86fc5f44, 2e039b84) and C1=(5fb01059, ea123520)
Running with n=4, C0=(86fc5f44, 2e039b84) and C1=(5fb01059, ea123520)
Running with n=4, C0=(86fc5f44, 2e039b84) and C1=(5fb01059, ea123520)
Running with n=4, C0=(86fc5f44, 2e039b84) and C1=(5fb01059, ea123520)
Dictionary size: 48B
Dictionary size: 48B
Dictionary size: 48B
Dictionary size: 48B
Dictionary size: 48B
Dictionary size: 48B
Dictionary size: 48B
Fill: 0.0s
Probe: 0.0s. 2 candidate pairs tested
Fill: 0.0s
Probe: 0.0s. 0 candidate pairs tested
Fill: 0.0s
Probe: 0.0s. 5 candidate pairs tested
Fill: 0.0s
SOLUTION FOUND!
Probe: 0.0s. 2 candidate pairs tested
Fill: 0.0s
Probe: 0.0s. 3 candidate pairs tested
Fill: 0.0s
Probe: 0.0s. 2 candidate pairs tested
Fill: 0.0s
```

Probe: 0.0s. 3 candidate pairs tested
Solution found: (4, 5) [checked OK]

input: mpirun -np 7 ./golden$_collision--n25--C025fd55cae2cfea79--C125acc6d9fa627309$
output:
Running with n=25, C0=(e2cfea79, 25fd55ca) and C1=(fa627309, 25acc6d9)
Running with n=25, C0=(e2cfea79, 25fd55ca) and C1=(fa627309, 25acc6d9)
Running with n=25, C0=(e2cfea79, 25fd55ca) and C1=(fa627309, 25acc6d9)
Running with n=25, C0=(e2cfea79, 25fd55ca) and C1=(fa627309, 25acc6d9)
Running with n=25, C0=(e2cfea79, 25fd55ca) and C1=(fa627309, 25acc6d9)
Running with n=25, C0=(e2cfea79, 25fd55ca) and C1=(fa627309, 25acc6d9)
Running with n=25, C0=(e2cfea79, 25fd55ca) and C1=(fa627309, 25acc6d9)
Dictionary size: 115.0MB
Dictionary size: 115.0MB
Dictionary size: 115.0MB
Dictionary size: 115.0MB
Dictionary size: 115.0MB
Dictionary size: 115.0MB
Dictionary size: 115.0MB
Fill: 2.5s
Fill: 2.5s
Fill: 2.5s
Fill: 2.5s
Fill: 2.5s
Fill: 2.5s
Fill: 2.5s
SOLUTION FOUND!
Probe: 5.1s. 4791621 candidate pairs tested
Probe: 5.1s. 4793072 candidate pairs tested
Probe: 5.1s. 4793662 candidate pairs tested
Probe: 5.1s. 4787758 candidate pairs tested
Probe: 5.1s. 4790465 candidate pairs tested
Probe: 5.1s. 4789791 candidate pairs tested
Probe: 5.1s. 4804687 candidate pairs tested
Solution found: (8778989, 863789) [checked OK]

input: mpirun -np 7 ./golden$_collision--n27--C0d8cb1683503a28af--C13d8336740897e44c$
output:
Running with n=27, C0=(503a28af, d8cb1683) and C1=(0897e44c, 3d833674)
Running with n=27, C0=(503a28af, d8cb1683) and C1=(0897e44c, 3d833674)
Running with n=27, C0=(503a28af, d8cb1683) and C1=(0897e44c, 3d833674)
Running with n=27, C0=(503a28af, d8cb1683) and C1=(0897e44c, 3d833674)
Running with n=27, C0=(503a28af, d8cb1683) and C1=(0897e44c, 3d833674)
Running with n=27, C0=(503a28af, d8cb1683) and C1=(0897e44c, 3d833674)
Running with n=27, C0=(503a28af, d8cb1683) and C1=(0897e44c, 3d833674)
Dictionary size: 460.2MB
Dictionary size: 460.2MB
Dictionary size: 460.2MB
Dictionary size: 460.2MB
Dictionary size: 460.2MB
Dictionary size: 460.2MB

Dictionary size: 460.2MB
Fill: 9.8s
Fill: 9.8s
Fill: 9.8s
Fill: 9.8s
Fill: 9.8s
Fill: 9.8s
Fill: 9.8s
SOLUTION FOUND!
Probe: 20.7s. 19158698 candidate pairs tested
Probe: 20.7s. 19164653 candidate pairs tested
Probe: 20.7s. 19171466 candidate pairs tested
Probe: 20.7s. 19186594 candidate pairs tested
Probe: 20.7s. 19182251 candidate pairs tested
Probe: 20.7s. 19168379 candidate pairs tested
Probe: 20.7s. 19182081 candidate pairs tested
Solution found: (126007987, 10504027) [checked OK]

MPI_Alltoallv

```
//creating the send buffer and appropriate information for MPI_Alltoallv

    int num_elmts_s_buff = 0;
    int size_s_buff = mpi_size;

    struct entry *send_buffer = calloc(mpi_size ,sizeof(struct entry));
    int *s_count = calloc(mpi_size ,sizeof(int));
    int *s_displacement = malloc(mpi_size *sizeof(int));
    for (int i = 0; i< mpi_size; i++){
        s_displacement[i] = i;
    }

    for (u64 x = proc_start_val; x < proc_end_val; x++) {
        u64 z = f(x);
        int batch_id = z % mpi_size;
        struct entry curr;
        curr.k = z;
        curr.v = x;
        filling_s_buff(&send_buffer, curr, s_displacement[batch_id]+
           s_count[batch_id], &num_elmts_s_buff, &size_s_buff);
        s_count[batch_id]++;
        s_displacement[batch_id]++;
    }

    struct entry *send_buffer_correct = calloc(num_elmts_s_buff ,sizeof(
       struct entry));

    for (int full = 0; full < num_elmts_s_buff; full++){
        send_buffer_correct[full] = send_buffer[full];
    }
```

```
29      free(send_buffer);
30
31      int *r_count = malloc(mpi_size *sizeof(int));
32      int *r_displacement = malloc(mpi_size *sizeof(int));
33
34      MPI_Alltoall( s_count, 1, MPI_INT, r_count, 1 , MPI_INT,
            MPI_COMM_WORLD);
35      //Remember to free receive_info at the end
36
37      int size_rec = 0;
38      for (int count = 0; count < mpi_size; count++){
39          r_displacement[count] = size_rec * sizeof(struct entry);
40
41          size_rec += r_count[count];
42          r_count[count] *= sizeof(struct entry); // we need this for the
                MPI_Alltoallv
43
44          s_count[count] *= sizeof(struct entry);
45          s_displacement[count] *= sizeof(struct entry);
46      }
47
48      struct entry *rec_buffer = calloc(size_rec ,sizeof(struct entry));
49
50      MPI_Alltoallv(send_buffer_correct, s_count, s_displacement, MPI_BYTE,
            rec_buffer, r_count, r_displacement ,MPI_BYTE, MPI_COMM_WORLD);
51
52      for (int filling = 0; filling < size_rec ; filling ++){
53          if(rec_buffer[filling].k != 0 || rec_buffer[filling].v != 0)
54              dict_insert(rec_buffer[filling].k, rec_buffer[filling].v);
55      }
56      free(send_buffer_correct);
57      free(rec_buffer);
58      free(r_count);
59      free(r_displacement);
60      free(s_count);
61      free(s_displacement);
62
63
64      MPI_Barrier(MPI_COMM_WORLD);
65
66
67  /*--------------------------*/
68  int num_elmts_s_buff_2 = 0;
69      int size_s_buff_2 = mpi_size;
70
71      struct entry *send_buffer_2 = calloc(mpi_size ,sizeof(struct entry));
72      int *s_count_2 = calloc(mpi_size ,sizeof(int));
73      int *s_displacement_2 = malloc(mpi_size *sizeof(int));
74      for (int i = 0; i< mpi_size; i++){
75          s_displacement_2[i] = i;
76      }
```

```
77
78
79      int nres = 0;
80      u64 ncandidates = 0;
81      for (u64 z = proc_start_val; z < proc_end_val; z++) {
82          u64 y = g(z);
83          int batch_id = y % mpi_size;
84          struct entry curr;
85          curr.k = y;
86          curr.v = z;
87          filling_s_buff(&send_buffer_2, curr, s_displacement_2[batch_id]+
                s_count_2[batch_id], &num_elmts_s_buff_2, &size_s_buff_2);
88          s_count[batch_id]++;
89          s_displacement[batch_id]++;
90      }
91
92      struct entry *send_buffer_correct_2 = calloc(num_elmts_s_buff_2 ,
            sizeof(struct entry));
93
94      for (int full = 0; full < num_elmts_s_buff_2; full++){
95          send_buffer_correct_2[full] = send_buffer_2[full];
96      }
97      free(send_buffer_2);
98
99      int *r_count_2 = malloc(mpi_size *sizeof(int));
100     int *r_displacement_2 = malloc(mpi_size *sizeof(int));
101
102     MPI_Alltoall( s_count_2, 1, MPI_INT, r_count_2, 1 , MPI_INT,
            MPI_COMM_WORLD);
103     //Remember to free receive_info at the end
104
105     int size_rec_2 = 0;
106     for (int count = 0; count < mpi_size; count++){
107         r_displacement_2[count] = size_rec_2 * sizeof(struct entry);
108         size_rec_2 += r_count_2[count];
109         r_count_2[count] *= sizeof(struct entry); // we need this for the
                MPI_Alltoallv
110         s_count_2[count] *= sizeof(struct entry);
111         s_displacement_2[count] *= sizeof(struct entry);
112     }
113
114     struct entry *rec_buffer_2 = calloc(size_rec_2 ,sizeof(struct entry));
115
116     MPI_Alltoallv(send_buffer_2, s_count_2, s_displacement_2, MPI_BYTE,
            rec_buffer_2, r_count_2, r_displacement_2 ,MPI_BYTE, MPI_COMM_WORLD
            );
117
118     u64 x[256];
119     for (u64 i = 0; i < size_rec_2; i++) {
120         u64 y = rec_buffer_2[i].k;
121         u64 z = rec_buffer_2[i].v;
```

```
122         if ((y != 0 || z != 0)) {
123             int nx = dict_probe(y, 256, x);
124             if (nx < 0) {
125                     printf("Process %d: Error in dict_probe\n", mpi_rank);
126                 continue;
127             }
128             ncandidates += nx;
129             for (int j = 0; j < nx; j++) {
130                 if (is_good_pair(x[j], z)) {
131                     #pragma omp critical
132                     {
133                         if (nres < maxres) {
134                             k1[nres] = x[j];
135                             k2[nres] = z;
136                             printf("SOLUTION FOUND!\n");
137                             nres++;
138                         }
139                     }
140                 }
141             }
142         }
143     }
144     free(send_buffer_correct_2);
145     free(rec_buffer_2);
146     free(r_count_2);
147     free(r_displacement_2);
148     free(s_count_2);
149     free(s_displacement_2);
```