Parallel Programming
(PPAR)

Sorbonne University

Direct Meet-in-the-Middle Attack Project

By

Hani Abdallah - (ID: 21400302)

Mathurin De Crecy - (ID: 21402875)

Contents

1 Approach

Our approach takes a structured strategy by breaking the problem into 4 major components:

1. Dictionary distribution.
2. Key and value generation.
3. Data Exchange.
4. Identifying the solution.

By separating these components, we can independently execute each stage across multiple cores, which increase scalability and reduces computation time.

2 Implementation

2.1 Dictionary Setup

In this section, individual cores perform their part of generating the dictionary jointly. This way, parallel processing is allowed, which helps to save memory as well as improves resource usage without sacrificing any data or functionalities.

Instead of assigning the entire dictionary of size N to a single core, we divide the global dictionary evenly across all the available cores. With work being equally divided, processes can be faster and more efficient without any dependencies, as follows:

```
    u64 N = 1ull << n; // Global dictionary size

    u64 localN = N / cores; //Local dictionary size
```

2.1.1 Distributed Initialization via OpenMP

The contribution on OpenMP into the dictionary initialization, since this process is done separately on each core, then we can benefit from threading the initialization for loop using OpenMP, thus we can reduce the time needed to initialize the local dictionary on each core, leading to faster overall setup:

```
void dict_setup(u64 size, struct entry **A)
{
    dict_size = size;
    char hdsize[8];
    human_format(dict_size * sizeof(struct entry), hdsize);
    printf("Dictionary size: %sB\n", hdsize);

    *A = malloc(sizeof(struct entry) * dict_size);
    if (*A == NULL)
        err(1, "impossible to allocate the dictionnary");
#pragma omp parallel for
    for (u64 i = 0; i < dict_size; i++)
        (*A)[i].k = EMPTY;
}
```

2.2 Key and value generation

Each core will then use the fill function which has been customized to allow more granular control over the dictionary setup and to increase parallelization potential.

This function includes the "dict_setup" that initialize the dictionary, "Keys generation using $f(x)$ that generate Keys and respective values" and "dict_insert" which allow us to insert the generated keys in the dictionary along side with their respective values.

2.2.1 Concurrent Key/Value generation via OpenMP

```
void fill(u64 rank, u64 localN, struct entry **A)
{
    dict_setup(1.125 * localN, A);

#pragma omp parallel for

    for (u64 x = rank * localN; x < (rank + 1) * localN; x++)
    {
        u64 z = f(x);
        dict_insert(z, x, *A);
    }
}
```

Note: It's not always beneficial to implement OpenMP since sometimes ensuring that the code is "Thread safe" by using atomic or critical sections will actually make the execution slower.

2.3 Data Exchange

Once each node have initialized it local dictionary, it will start going over its (key,value) tuples, generating the "Keys generation using $g(x)$ " packing them into an 1D array of $(g(x), x)$. All the other nodes, will do exactly the same as these values with be shared between all nodes.

There might exist multiple strategy on how to share these values using MPI, but as we noticed that each node must share its value to all other nodes, the most natural and straight forward operation was $MPI\_ALLGATHER()$, this operation shine as it guarantee consistency and integrity of the shared data across all the processors of the communicator, since $MPI\_ALLGATHER()$ is a collective operation.

2.3.1 Exchange Mechanism via MPI

```
for (int x = 0; x < limitloop - n; x += n)
    {
        for (int i = 0; i < 2 * n; i += 2)
        {
            int q = x + (i / 2);

            if (A[q].k != EMPTY && q < limitloop){

                y[i] = g(A[q].v);
                y[i + 1] = A[q].v;

                } else
                {

                y[i] = 0;
                y[i + 1] = 1;
            }
        }
    MPI_Allgather(&y, 2 * n, MPI_UINT64_T, potential_solutions, 2 * n, MPI_UINT64_T,MPI_COMM_WORLD);

        }
```

2.4 Identifying the solution

After exchanging the generated values between all nodes, each node will iterate individually over its own generated values and the received ones with its own local dictionary that was previously initialized and evaluate them to find a solution.

The seek() function, is a custom function that reuse the already provided isgoodpair() function to check these values.

```
for (int j = 0; j < 2 * n * cores; j += 2)
{
    if (potential_solutions[j] != 0)
    {
        seek(potential_solutions[j], potential_solutions[j + 1], res, A, rank);
        if (res[0] != 0 || res[1] != 0)
        {
            solved = 1;
            break;
        }
    }
}
```

2.5 Optimization

2.5.1 Number of nodes

After analyzing our parallel code on different node numbers as its shown in section "3.2 Efficient number of cores", it was clear that the best choice of the number of nodes is 2^n (power of 2), since as mentioned in section "2.1 Dictionary Setup", the *Global dictionary* size is 2^n.

But note that choosing the highest number of nodes is not always the best option, as the cost of communication will then increase and become more than the cost of processing and memory organization.

2.5.2 Auto-vectorization

By using Optimization level 3, we can implement Auto-vectorization on our parallel code, along side with other optimizations done by the compiler, below are the vectorized code blocks, generated by the vectorization report:

```
~$ mpicc -O3 -fopenmp -fopt-info-vec mitm.c -o run
mitm.c:240:9: optimized: basic block part vectorized using 16 byte vectors
mitm.c:256:5: optimized: basic block part vectorized using 16 byte vectors
mitm.c:274:24: optimized: basic block part vectorized using 16 byte vectors
mitm.c:400:26: optimized: basic block part vectorized using 16 byte vectors
```

2.6 Bottleneck

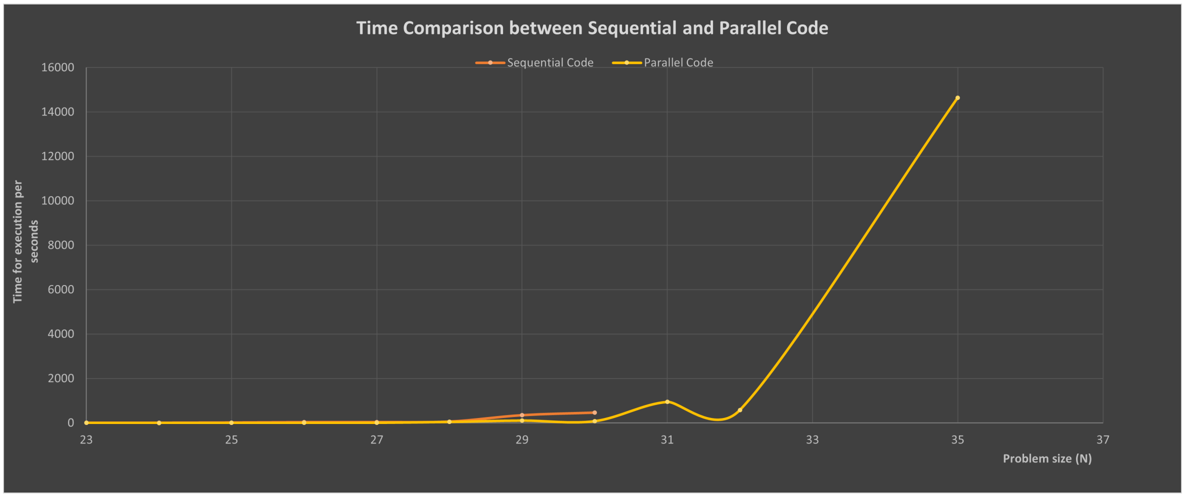
Choosing the right number of nodes to execute the parallel code is crucial, as we must take a decision to choose between better space handling or faster execution time.

Choosing a slow network infrastructure will affect the execution time, as the problem size N increases, the memory required to store the dictionaries increases, this will then becomes a major bottleneck as the number of communication between the nodes will massively increase as well, so having a faster network is highly recommended.

3 Performance Analysis

3.1 Sequential vs Parallel code

MITM Parallelization potential		
Problem Size (N)	Sequential code (per seconds)	Parallel code (per seconds)
23	1.93	1.33
24	4.25	0.09
25	14.04	2.52
26	32.06	3.36
27	34.79	9.63
28	56.99	47.63
29	346.53	109.8
30	461.76	81.85
31	N/A	948.62
32	N/A	585.17
35	N/A	14642.51



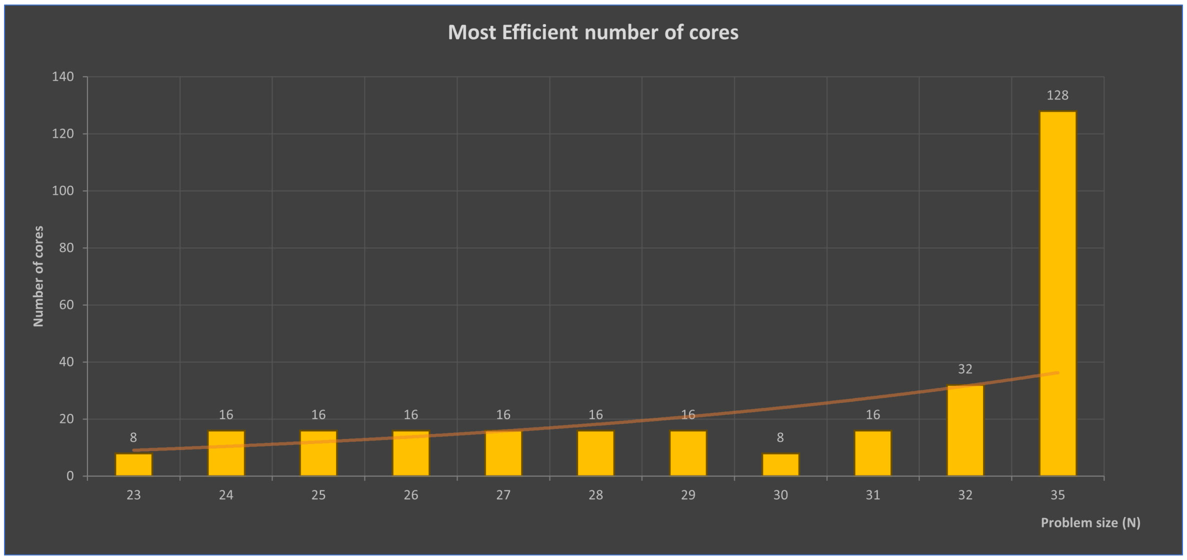
In order to have more insights on he exact performance of our parallel code, we will be analyzing the **Speed-up**, **Efficiency** and **Performance Improvement**.

But we can only do this comparison with problem size n ¡= 30, and that's because we will couldn't know the execution time needed by the sequential code to finish.

Efficient number of cores			
Problem Size (N)	Speed-up (s)	Efficiency (s)	Performance Improvement (%)
23	1.45	0.181	31.09
24	47.22	2.951	97.88
25	5.57	0.348	82.05
26	9.54	0.596	89.52
27	3.61	0.226	72.32
28	1.20	0.075	16.42
29	3.16	0.197	68.31
30	5.64	0.705	82.27

3.2 Efficient number of cores

Efficient number of cores	
Problem Size (N)	Number of cores
23	8
24	16
25	16
26	16
27	16
28	16
29	16
30	8
31	16
32	32
35	128



4 Results

Here, we will share our testing results for the parallel code :

MITM Solutions			
Problem Size (N)	Solution	Number of cores	Execution time (s)
23	(5dfc66, 23e131)	8	1.33
24	(f38828, 54316e)	16	0.09
25	(181ddec, 19d2b2f)	16	2.52
26	(27d1996, 3a9ea5e)	16	3.36
27	(77187ab, 1b40e08)	16	9.63
28	(ddaa0d2, 3a16db)	16	47.63
29	(453dc1c,1e01a07f)	16	109.8
30	(16fbcb2f,15886bdc)	8	81.85
31	(336ffdc3,4b2a749a)	16	948.62
32	(7c7d3242,e75269e1)	32	585.17
35	(2a2b17f7e,9801d87)	128	14642.51

The username used to choose the challenges is : ”parallel”.
https://ppar.tme-crypto.fr/parallel/35