Sorbonne Université
MU4IN903 (PPAR)

Direct Meet-in-the-Middle Attack Project

Jan ESQUIVEL MARXEN, Thomas GANTZ

January 16, 2025

Contents

1 Implementation

Before delving into the details of the implementation, we summarize the preliminary considerations and challenges that shaped our approach:

- The proposed algorithm offers a significant improvement in speed over the brute-force method. However, it introduces a bottleneck in terms of memory usage due to the need to store a large dictionary. To address this, distributed machine parallelism with sharding is essential for handling large values of n.
- At the highest level, the algorithm operates in two phases:
  1. Filling the dictionary.
  2. Searching within the dictionary.

  These phases are suitable for Message Passing Interface (MPI) parallelization. For inner loops, OpenMP can be used to exploit thread-level parallelism, while Advanced Vector Extensions (AVX) can accelerate function evaluations.
- A challenge arises in determining which process stores a specific key, as this depends on the value of $f(x)$, which is not known *a priori*.
- Another issue is the variability in the size of messages exchanged between processes. To handle this, a two-step communication pattern is necessary: first, sending the size of the data, followed by the actual data payload.
- Load balancing is not a concern in this context because the functions f, g, and π exhibit consistent computation times regardless of the input data. Consequently, a static block distribution suffices, eliminating the need for dynamic or cyclic workload distribution strategies.

1.1 Minimal MPI (v1)

Our initial strategy focused on implementing the simplest MPI-based approach to construct a distributed dictionary.

1.1.1 Dictionary Distribution

To achieve this, we partitioned the first loop:

```c
for (u64 x = (my_rank * N) / p; x < ((my_rank + 1) * N
    ) / p; x++) {
    u64 z = f(x);
    dict_insert(z, x);
}
```

As a result, the dictionary is distributed across the p participating processes. While this approach is functional, it introduces inefficiencies. Specifically, during the dictionary lookup phase, each process does not inherently know which process holds a given key. Consequently, in the second lookup loop, every process must test all values of $g(y)$ to determine whether they exist in its local dictionary. Despite this inefficiency, this approach serves as a solid starting point.

1.1.2 Solution Exchange

To collect the solution(s), we implemented a gathering mechanism at the root process:

```c
int *global_nres = NULL;  // Array to store the number
     of results from each process
int *displs = NULL;       // Array to store
    displacement offsets for gathered data

// Root process (rank 0) allocates memory for the
    global result count and displacements
if (my_rank == 0) {
    global_nres = malloc(p * sizeof(int));
    displs = malloc(p * sizeof(int));
}

// Gather the number of results (nres) from all
    processes to the root process
MPI_Gather(&nres, 1, MPI_INT, global_nres, 1, MPI_INT,
     0, MPI_COMM_WORLD);
```

```c

int total_nres = 0;  // Total number of gathered results

// Root process calculates displacements and total size of gathered data
if (my_rank == 0) {
    displs[0] = 0;
    for (int i = 0; i < p; i++) {
        total_nres += global_nres[i];  // Sum up all received results
        if (i > 0) {
            displs[i] = displs[i - 1] + global_nres[i - 1];  // Compute displacement offsets
        }
    }

    // Allocate global arrays for collected k1 and k2 values
    *K1 = malloc(total_nres * sizeof(u64));
    *K2 = malloc(total_nres * sizeof(u64));
}

// Gather k1 and k2 arrays from all processes into K1 and K2 on the root process
MPI_Gatherv(k1, nres, MPI_UNSIGNED_LONG_LONG, *K1, global_nres, displs,
            MPI_UNSIGNED_LONG_LONG, 0, MPI_COMM_WORLD);
MPI_Gatherv(k2, nres, MPI_UNSIGNED_LONG_LONG, *K2, global_nres, displs,
            MPI_UNSIGNED_LONG_LONG, 0, MPI_COMM_WORLD);

// Root process prints the gathered results
if (my_rank == 0) {
    printf("Total results gathered: %d\n", total_nres);
    for (int i = 0; i < total_nres; i++) {
        printf("K1[%d] = %lu, K2[%d] = %lu\n", i, *K1[i], i, *K2[i]);
    }
}
```

1.1.3 Main Adaptation

Finally, we modified the main function accordingly:

```c
#include <mpi.h>

int main(int argc, char **argv) {
    MPI_Init(NULL, NULL);  // Initialize the MPI environment
    ...
    int my_rank, p;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  // Get the rank of the current process
    MPI_Comm_size(MPI_COMM_WORLD, &p);        // Get the total number of processes
    printf("p=%d\n", p);

    // Setup dictionary with an approximate size (not necessarily divisible by p)
    dict_setup((u64)(1.5 * (1ull << n) / p));

    /* Begin search process */
    u64 *K1, *K2;  // Arrays to store search results
    double start_time, end_time;
    start_time = MPI_Wtime();
    int nkey = golden_claw_search(16, &K1, &K2, my_rank, p);
    end_time = MPI_Wtime();
    printf("Rank %d: Time taken = %f seconds\n", my_rank, end_time - start_time);

    if (my_rank == 0) {  // Root process handles validation
        assert(nkey > 0);  // Ensure at least one key was found

        /* Validation of results */
        for (int i = 0; i < nkey; i++) {
            printf("Validation step %d:\n", i);
            ...
            // Print validated solution
            printf("Solution found: (%" PRIx64 ", %" PRIx64 ") [checked OK]\n",
                   K1[i], K2[i]);
        }
    }
    MPI_Finalize();  // Finalize the MPI environment
}
```

1.2 All-to-All MPI (v2)

Building upon the previous approach, the goal in this version was to partition the second loop as well.

1.2.1 Systematic Dict Distribution

To achieve this partitioning, we needed a systematic distribution of keys to ensure that each process knows which keys might be present in its local dictionary. The distribution follows the rule: process i holds all keys for which `key` satisfies `key` $\mod i == 0$.

The implementation proceeds in two phases:

1. Each process computes $f(x)$ for its assigned block of x-values and determines the appropriate recipient process based on the distribution rule. The processes then exchange this data. After this exchange, each process inserts the received keys into its local dictionary.

2. Each process computes $g(y)$ for its assigned block of y-values and determines the responsible recipient. Only the designated process needs to check for the presence of $g(y)$ in its local dictionary. Therefore these values of $g(y)$ will be sent to him.

```c
/* STEP 1: Build partitioned dictionary.
 * Logical step: Process i holds all keys where (key % p == i). */

struct u64_darray *dict_zx = malloc(p * sizeof(struct u64_darray));

// Initialize dictionaries for each process with estimated capacity
for (int rank = 0; rank < p; rank++) {
    initialize_u64_darray(&(dict_zx[rank]), 2 * N / (p * p));
}

// Distribute values across processes
for (u64 x = (my_rank * N) / p; x < ((my_rank + 1) * N) / p; x++) {
    u64 z = f(x);             // Compute function f(x)
    int dest_rank = z % p;    // Determine the responsible process
    append(&(dict_zx[dest_rank]), z);  // Store z-value
    append(&(dict_zx[dest_rank]), x);  // Store corresponding x-value
}

// Exchange dictionary data between processes
u64 *dict_zx_recv;
int dict_zx_recv_size = exchange(&dict_zx_recv, dict_zx, p, my_rank);
free(dict_zx);

// Insert received values into local dictionary
for (int i = 0; i < dict_zx_recv_size - 1; i += 2) {
    dict_insert(dict_zx_recv[i], dict_zx_recv[i + 1]);
}
free(dict_zx_recv);

/* STEP 2: Attribute zs to their respective processes.
 * Logical step: Each process looks for zs in the dictionary of the process
 * that could potentially contain it (dest_rank = z % p). */

struct u64_darray *dict_zy = malloc(p * sizeof(struct u64_darray));

```

```c
// Initialize dynamic arrays for distributing g(y) results
for (int rank = 0; rank < p; rank++) {
    initialize_u64_darray(&(dict_zy[rank]), 2 * N / (p * p));
}

// Distribute g(y) values to respective processes
for (u64 y = (my_rank * N) / p; y < ((my_rank + 1) * N) / p; y++) {
    u64 z = g(y);            // Compute function g(y)
    int dest_rank = z % p;   // Determine responsible process
    append(&(dict_zy[dest_rank]), z);  // Store z-value
    append(&(dict_zy[dest_rank]), y);  // Store corresponding y-value
}

// Exchange dictionary data between processes
u64 *dict_zy_recv;
int dict_zy_recv_size = exchange(&dict_zy_recv, dict_zy, p, my_rank);
free(dict_zy);

/* STEP 3: Search for matches in the local dictionary. */
...
// Iterate through received z-y pairs and search in local dictionary
for (int i = 0; i < dict_zy_recv_size; i += 2) {
    u64 z = dict_zy_recv[i];
    u64 y = dict_zy_recv[i + 1];
    ...
}
free(dict_zy_recv);
```

1.2.2 Dynamic Arrays

Since the exact number of values to be sent to each process is unknown beforehand, we use an array of dynamic arrays (defined in `utilities.h`), each initialized with the expected capacity of $\frac{N}{p^2}$. Given a balanced workload, each process contributes $\frac{N}{p}$ elements, redistributed among p destinations, leading to an expected $\frac{N}{p^2}$ elements per destination. This estimate balances memory efficiency and reallocations despite variations.

1.2.3 Exchange Mechanism

To facilitate data exchange, we implemented a exchange function in `communication.h`, as this pattern occurs twice in the algorithm. The function utilizes either immediate `MPI_Isend`/`MPI_Irecv` calls or the `MPI_Alltoall` operation to avoid deadlocks.

```c
int exchange(u64 **recv, struct u64_darray *send, int p, int my_rank) {
    /* STEP 1: Communicate send sizes to other processes.
     * a) Each process sends the size of its data to all other processes.
     * b) Compute contiguous send displacements.
     */

    MPI_Request size_request[p];
    int send_sizes[p];          // Array storing data sizes to send to each rank
    int send_total_size = 0;    // Total size of data to be sent
    int sender_packet[2 * (p - 1)]; // Stores process rank and message size pairs
    int i = 0;

    // Loop through each destination process to send size information
    for (int dest_rank = 0; dest_rank < p; dest_rank++) {
        if (dest_rank == my_rank) continue;
        sender_packet[i] = my_rank;                   // Sender process rank
        sender_packet[i + 1] = send[dest_rank].size;  // Number of elements to send
        MPI_Isend(sender_packet + i, 2, MPI_INT, dest_rank, 0,
                  MPI_COMM_WORLD, &(size_request[dest_rank]));

        send_sizes[dest_rank] = send[dest_rank].size;
        send_total_size += send_sizes[dest_rank];
        i += 2;
    }

    // Include the local process's own send size
    send_sizes[my_rank] = send[my_rank].size;
    send_total_size += send_sizes[my_rank];

    // Compute displacements for contiguous memory allocation
    int send_displacements[p];
    send_displacements[0] = 0;
    for (int dest_rank = 1; dest_rank < p; dest_rank++) {
        send_displacements[dest_rank] = send_displacements[dest_rank - 1] + send_sizes[dest_rank - 1];
    }

    /* STEP 2: Receive size information from other processes.
     * a) Each process receives message sizes from all other processes.
     * b) Compute contiguous receive displacements.
     * c) Allocate memory for received data.
     */

    MPI_Request recv_size_request;
    int recv_sizes[p];          // Array storing received data sizes
    int recv_total_size = 0;    // Total size of data to be received

    // Loop to receive size information from each sender
    for (int sender = 0; sender < p; sender++) {
        if (sender == my_rank) continue;
        int recv_packet[2]; // Stores sender process rank and message size
        MPI_Irecv(recv_packet, 2, MPI_INT, MPI_ANY_SOURCE, 0,
                  MPI_COMM_WORLD, &recv_size_request);
        MPI_Wait(&recv_size_request, MPI_STATUS_IGNORE);
        recv_sizes[recv_packet[0]] = recv_packet[1];
        recv_total_size += recv_packet[1];
    }

    // Include the local process's own receive size
    recv_sizes[my_rank] = send[my_rank].size;
    recv_total_size += recv_sizes[my_rank];

    // Compute displacements for contiguous receive allocation
    int recv_displacements[p];
    recv_displacements[0] = 0;
    for (int sender = 1; sender < p; sender++) {
        recv_displacements[sender] =
            recv_displacements[sender - 1] + recv_sizes[sender - 1];
    }

    /* STEP 3: Pack data into contiguous memory and perform MPI communication.
     * a) Flatten dynamically allocated arrays into a single contiguous array.
     * b) Perform MPI_Alltoallv to exchange the data.
     * c) Allocate and receive the data in a contiguous buffer.
     */

    // Allocate contiguous memory for sending data
    u64 *contiguous_send = malloc(send_total_size * sizeof(u64));

    // Parallelized memory copy operation
#pragma omp parallel for
    for (int sender = 0; sender < p; sender++) {
        memcpy(contiguous_send + send_displacements[sender],
               send[sender].data, send[sender].size * sizeof(u64));
        free_u64_darray(&(send[sender])); // Free original arrays after copying
    }

    // Allocate memory for receiving data with proper alignment for vectorization
    *recv = aligned_alloc(32, sizeof(u64) * recv_total_size);

    // Perform all-to-all variable-sized communication
    MPI_Alltoallv(contiguous_send, send_sizes, send_displacements,
                  MPI_UINT64_T, *recv, recv_sizes, recv_displacements,
                  MPI_UINT64_T, MPI_COMM_WORLD);

    free(contiguous_send);
    return recv_total_size;
}
```

1.3 OpenMP (v3)

Building upon version 2, we introduced OpenMP to leverage shared-memory parallelism. The most significant performance improvements may be achieved in two key areas.

1.3.1 Dictionary Probing

The dictionary probing phase was parallelized by distributing the lookup operations among multiple threads. While deadlocks were no longer a concern, race conditions became an issue due to shared-memory access. Therefore, all potentially unsafe operations (i.e., those involving at least one write to a shared variable) had to be properly synchronized.

The two critical cases were:

- Updating `nx`: This update was made thread-safe using an atomic operation.
- Appending solutions: Since solutions are rarely found, appending to the solution array was placed within a critical region. The impact on performance is minimal, as threads will rarely need to enter this region.

Additionally, we encountered a technical constraint: OpenMP parallel regions do not allow `return` or `break` statements. To handle this, we introduced a flag that signals termination, with the actual return occurring after the parallel region. This flag did not require explicit synchronization since race conditions in setting it do not affect correctness—once set, it remains set.

```
#include <omp.h>

int nres = 0;        // Number of valid (k1, k2) pairs found
u64 ncandidates = 0; // Number of candidate values probed
u64 k1[16], k2[16];  // Arrays to store found (k1, k2) pairs
int flag = 0;        // Flag to stop iterations once max results are reached

#pragma omp parallel shared(flag, nres, ncandidates, k1, k2)
{
    // Temporary buffer to store probed x values
    u64 x[256 / omp_get_num_threads()];

    #pragma omp for
    for (int i = 0; i < dict_zy_recv_size; i += 2) {
        // Probe dictionary for matches
        u64 z = dict_zy_recv[i];
        u64 y = dict_zy_recv[i + 1];
        int nx = dict_probe(z, 256 / omp_get_num_threads(), x);
        assert(nx >= 0);

        #pragma omp atomic
        ncandidates += nx; // Atomically update candidate count

        for (int i = 0; i < nx; i++) {
     if (is_good_pair(x[i], y)) {
      if (flag) continue;
                if (nres < maxres) {
       printf("SOLUTION FOUND! by thread %d\n", omp_get_thread_num());

                #pragma omp critical
      { // Ensure protected updates to shared arrays
                    k1[nres] = x[i];
                    k2[nres] = y;
                    nres++;
      }
      } else {
       flag = 1; // Set flag to stop further iterations
      }
            }
 }
   }
}
if (flag) return -1;
free(dict_zy_recv);
```

1.3.2 Appending to Dynamic Arrays

The second optimization involved parallelizing the distribution of $f(x)$ values for insertion and $g(y)$ values for probing. Since these values are stored in an array of dynamic arrays, concurrent writes had to be synchronized.

To achieve this, we used locks per destination process to ensure safe appends. Using a single critical region for all appends would have been inefficient, as different threads often write to different arrays simultaneously.

```
omp_lock_t locks[p];
for (int i = 0; i < p; i++) {
    omp_init_lock(&locks[i]);
}

#pragma omp parallel
{
    #pragma omp for
    for (u64 x = (my_rank * N) / p; x < ((my_rank + 1) * N) / p; x++) {
        u64 z = f(x);
        int dest_rank = z % p;

        // Lock the destination rank's dictionary to avoid race conditions
        omp_set_lock(&locks[dest_rank]);
        append(&(dict_zx[dest_rank]), z);
        append(&(dict_zx[dest_rank]), x);
        omp_unset_lock(&locks[dest_rank]);
    }
}

for (int i = 0; i < p; i++) {
    omp_destroy_lock(&locks[i]);
}
```

1.3.3 Thread Usage

Finally, we experimented with the number of threads to maximize efficiency.

```
    omp_set_num_threads(20);
```

On the Nancy's `gros` cluster (later used in the evaluation), each node provides 18 physical cores, each supporting two logical threads, totaling 36 hardware threads. However, using all 36 threads was inefficient in practice.

1.4 AVX2 (v4)

In this version, built upon version 3, we implemented a vectorized approach using AVX2, since we limited our computations to clusters supporting only this AVX version.

1.4.1 Murmur Hash Optimization

We observed that the Murmur hash function performs a sequence of basic calculations for each key lookup and insertion. Since we process multiple keys at once, these operations are well-suited for vectorization. By using AVX2, we parallelized these computations to improve throughput.

```c
#include <immintrin.h>

__m256i mul1 = _mm256_set1_epi64x(0
     xff51afd7ed558ccdull);
__m256i mul2 = _mm256_set1_epi64x(0
     xc4ceb9fe1a85ec53ull);

__m256i murmur64_avx2(__m256i x_vec) {
    // First mixing step: x ^= (x >> 33)
    __m256i x_shifted = _mm256_srli_epi64(x_vec, 33);
    x_vec = _mm256_xor_si256(x_vec, x_shifted);
    // Multiply by first constant
    x_vec = avx2_mullo_epi64(x_vec, mul1);
    // Second mixing step: x ^= (x >> 33)
    x_shifted = _mm256_srli_epi64(x_vec, 33);
    x_vec = _mm256_xor_si256(x_vec, x_shifted);
    // Multiply by second constant
    x_vec = avx2_mullo_epi64(x_vec, mul2);
    // Final mixing step: x ^= (x >> 33)
    x_shifted = _mm256_srli_epi64(x_vec, 33);
    x_vec = _mm256_xor_si256(x_vec, x_shifted);
    return x_vec;
}
```

1.4.2 Parallelized Insertions

We applied AVX2 specifically to parallelize key insertions. To ensure efficient vectorized operations, data alignment was enforced to 32 bytes in the exchange function (see above). For handling any remaining elements that did not fit into full vectorized batches, we used the standard scalar insertion function as an epilogue.

```c
for (int i = 0; i < dict_zx_recv_size; i += 8) {
    // Load 4 keys and 4 values into AVX2 registers
    __m256i keys = _mm256_set_epi64x(
        dict_zx_recv[i + 6], dict_zx_recv[i + 4],
        dict_zx_recv[i + 2], dict_zx_recv[i]);  //
             Load keys

    __m256i values = _mm256_set_epi64x(
        dict_zx_recv[i + 7], dict_zx_recv[i + 5],
        dict_zx_recv[i + 3], dict_zx_recv[i + 1]);  //
             Load values

    dict_insert_avx2(keys, values);
}

// Epilogue: process remaining elements
for (int i = dict_zx_recv_size - dict_zx_recv_size %
     8;
     i < dict_zx_recv_size; i += 2) {
 dict_insert(dict_zx_recv[i], dict_zx_recv[i + 1]);
}
```

1.4.3 AVX2 Insertion Function

The dictionary insertion function is structured to leverage parallelism where possible. While hash key computation can be efficiently vectorized, the actual insertion into the dictionary remains inherently sequential due to potential collisions and memory updates.

```c
void dict_insert_avx2(__m256i keys, __m256i values) {
    // Compute vectorized Murmur64 hashes
    __m256i hashes = murmur64_avx2(keys);

    for (int i = 0; i < 4; i++) {
        // Extract individual hash and compute initial
             index
        u64 h = avx2_extract_epi64(hashes, i) %
             dict_size;

        // Linear probing for empty slot
        for (;;) {
            if (A[h].k == EMPTY) break;
            h += 1;
            if (h == dict_size) h = 0;  // Wrap around
        }

        assert(A[h].k == EMPTY);

        // Insert key and value
        A[h].k = avx2_extract_epi64(keys, i) % PRIME;
        A[h].v = avx2_extract_epi64(values, i);
    }
}
```

1.4.4 AVX2 Helper Functions

Since AVX2 lacks certain intrinsic operations that would be useful for our implementation, we manually implemented helper functions to emulate some AVX-512 functionality in software.

```c
// Extract 64-bit integer from __m256i at position `
     index`
static inline u64 avx2_extract_epi64(__m256i vec, int
     index) {
    __m128i low = _mm256_castsi256_si128(vec);    //
         Lower 128 bits
    __m128i high = _mm256_extracti128_si256(vec, 1);
         // Upper 128 bits

    switch (index) {
        case 0: return _mm_extract_epi64(low, 0);
        case 1: return _mm_extract_epi64(low, 1);
        case 2: return _mm_extract_epi64(high, 0);
        case 3: return _mm_extract_epi64(high, 1);
        default: return 0;  // Invalid index
    }
}

// Emulate _mm256_mullo_epi64 using AVX2
__m256i avx2_mullo_epi64(__m256i a, __m256i b) {
    // Extract lower and higher 32-bit parts
    __m256i a_lo = _mm256_and_si256(a,
         _mm256_set1_epi64x(0xFFFFFFFF));
    __m256i a_hi = _mm256_srli_epi64(a, 32);
    __m256i b_lo = _mm256_and_si256(b,
         _mm256_set1_epi64x(0xFFFFFFFF));
    __m256i b_hi = _mm256_srli_epi64(b, 32);

    // Perform 32-bit multiplications
    __m256i lo_lo = _mm256_mul_epu32(a, b);     // Low
          32 bits * Low 32 bits
    __m256i hi_lo = _mm256_mul_epu32(a_hi, b);  //
         High 32 bits * Low 32 bits
    __m256i lo_hi = _mm256_mul_epu32(a, b_hi);  // Low
          32 bits * High 32 bits

    // Combine results
    __m256i result = _mm256_add_epi64(lo_lo,
         _mm256_slli_epi64(hi_lo, 32));
    result = _mm256_add_epi64(result,
         _mm256_slli_epi64(lo_hi, 32));

    return result;
}
```

2 Evaluation

We evaluate our different parallelization schemes using **weak scaling**.

2.1 Setup

For all computations, we have used the Grid5000 network. Specifically, we used Nancy's `gros` cluster. Since all nodes in the cluster have the same specifications except for slight variations in SSD storage, we have chosen the nodes at random to get resources faster. For details on the cluster, refer to the Grid5000 Nancy Hardware page.

2.2 Results and Discussion

2.2.1 Weak Scaling

In this section weak scaling results for all versions are presented. The key size n is incremented by 1 from 25 to 30, thus doubling the dictionary size at every increment. The script automating this can be consulted in Appendix A.1. All resource reservations were done in the following way: `oarsub -p "gros" -l host=<max amount of MPI processes>/core=18,walltime=<max expected execution time> -O <output file> -E <error file> <bash script to run>`. For high resource requests nightly hours were chosen using the -r parameter. We have chosen OpenMP threads for versions 3 and 4 to be around 20 (not the full 36 available logical cores). The table and graph show the execution time increase.

Problem Size	Nodes Used	Original	Version 1	Version 2	Version 3	Version 4
25	1	19	16	20	15	15
26	2	37	22	20	16	16
27	4	76	35	21	39	53
28	8	162	62	24	42	57
29	16	351	129	30	56	69
30	32	802	251	36	61	69

Table 1: Weak scaling results for multiple versions. The table shows total execution times in seconds for different problem sizes and versions of the program.

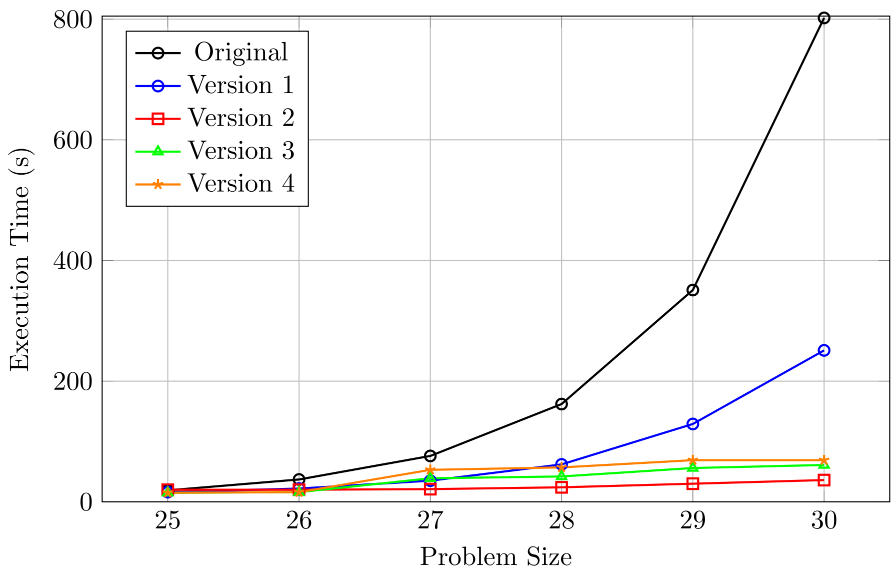

Figure 1: Execution time vs. problem size for different versions. The plot compares the performance of the versions across multiple problem sizes.

Given that the dictionary size, $N = 2^n$, doubles with each increment, as well as the number of computing nodes, p, the amount of data to be processed by each node (or MPI process), $\frac{N}{p}$, remains constant. If no communication were required, the total execution time would also remain constant, as the workload per node is unchanged. However, communication between processes is unavoidable, and its cost increases with the number of nodes. Specifically, the communication overhead grows as $\mathcal{O}(p^2)$, leading to an increase in execution time as the number of nodes increases. This explains the observed growth in execution time despite the constant amount of data per node.

With two network interface controllers, the cluster can potentially achieve an aggregate bandwidth of up to 50 Gbps (3.125 Gb/s). However, the network may approach its bandwidth limits as more nodes participate in the communication, causing congestion.

The sudden jumps in execution time can be attributed to several factors. MPI implementations often buffer messages to optimize throughput and reduce latency, which consumes additional memory and introduces memory overhead. As n increases, the additional message streams may exceed the buffering or memory bandwidth capacity, leading to more frequent stalls and thus an increase in execution time. Further investigation is required to fully understand the underlying causes of these performance fluctuations.

2.2.2 Stress Test

Values higher than $n = 31$ were tested (see Appendix A.2) but were unsuccessful, as processes were terminated from this size onwards. After investigating with tools like `valgrind` and `massif`, we confirmed that peak memory usage per processor was not the issue. Instead, the problem arises within our primary MPI operation, `MPI_Alltoallv()`. This suggests that the size of the data might be too large to be efficiently transferred across the cluster network, or the MPI operation itself could be allocating excessive temporary storage in some processes.

To address these challenges, several approaches could be explored:

- **Batched Processing**: Divide the data into smaller batches that fit into each CPU's cache and can more easily be handled by the network.
- **MPI-Specific Tuning**: Adjust MPI configuration settings, such as decreasing buffer sizes.
- **Specialized Hardware**: Use clusters with higher network bandwidth or specialized interconnects (e.g., InfiniBand) that can handle larger data sizes more effectively.
- **Algorithm Rethink**: Study if restructuring the algorithm could further reduce communication.

3 Conclusion

The primary objective of parallelizing computations across multiple nodes using MPI was successfully achieved. This enabled the program to scale effectively across many nodes.

However, the integration of OpenMP did not yield significant performance improvements. The overhead associated with threading outweighed the computational work assigned to each thread during the lookup operations. Reducing the number of threads could potentially reduce this overhead and improve performance, as it would decrease the cost of thread creation and synchronization.

Similarly, the inclusion of AVX2 vectorization failed to improve performance, as the cost of load/store operations on vector registers surpassed the benefits of the computations performed. Additionally, the necessity to emulate AVX512 functions introduced further inefficiencies. It is plausible that using hardware with native AVX512 support could have led to a different outcome and potentially some performance gains.

Unfortunately, we were only able to compute up to $n = 31$, with solution (5d0c8888, 50041dbc) and username: jan.marxen. Nonetheless, the suggestions outlined in the evaluation section provide clear pathways for overcoming these limitations. Future work addressing these could easily scale the application beoyond $n = 31$.

A Evaluation Scripts

This appendix provides the content of the shell scripts used in the project. These scripts automate the execution of weak scaling and stress tests.

A.1 `version_comparison.sh`

The `version_comparison.sh` script automates weak scaling experiments for various versions. Its full content is shown below:

```
#!/bin/bash

# Set the base URL
BASE_URL="https://ppar.tme-crypto.fr/jan.marxen/"

# Range of problem sizes
PROBLEM_SIZES=(25 26 27 28 29 30)  # You can modify this range as needed

# Versions to test
VERSIONS=("original" "version1" "version2" "version3" "version4")  # Define all versions here

# Loop through each problem size
nodes=1

# Prepare the weak_scaling.txt file to store the results
echo "Problem Size, Number of Nodes, ${VERSIONS[@]}" > versions_weak_scaling.txt

for problem_size in "${PROBLEM_SIZES[@]}"; do
    # Download the text file for the current problem size
    URL="${BASE_URL}${problem_size}"
    wget -q -O challenge.txt "$URL"

    # Extract parameters n, C0, and C1 from the text file
    n=$(grep -oP "(?<=--n )\d+" challenge.txt)

    # Extract C0 and C1 correctly (handle the two elements in each tuple)
    C0=$(grep -oP "(?<=C0 = \().*(?=\))" challenge.txt | tr -d '[:space:]')
    C1=$(grep -oP "(?<=C1 = \().*(?=\))" challenge.txt | tr -d '[:space:]')

    # Check if the values are extracted properly
    if [ -z "$n" ] || [ -z "$C0" ] || [ -z "$C1" ]; then
        echo "Failed to extract parameters for problem size $problem_size."
        continue
    fi

    # Split C0 and C1 into their respective elements (using comma as delimiter)
    C0_SECOND=$(echo $C0 | cut -d',' -f1)
    C0_FIRST=$(echo $C0 | cut -d',' -f2)
    C1_SECOND=$(echo $C1 | cut -d',' -f1)
    C1_FIRST=$(echo $C1 | cut -d',' -f2)

    # Print the extracted parameters for verification
    echo "Problem size: $problem_size"
    echo "n: $n, C0: ($C0_FIRST, $C0_SECOND), C1: ($C1_FIRST, $C1_SECOND)"

    # Create hostfile
    INPUT_FILE="$OAR_NODE_FILE"
    HOST_FILE="mpi_hostfile"

    # Deduplicate and count slots for each node
    awk '{slots[$1]++} END {for (host in slots) print host
    " slots=" slots[host]}' "$INPUT_FILE" > "$HOST_FILE"
    # Initialize a string to hold execution times for the current problem size
    EXEC_TIMES=""

    # Inner loop to run each version and measure execution time
    for VERSION in "${VERSIONS[@]}"; do
        # Run the MPI program and time it
        START_TIME=$(date +%s)
        mpiexec --hostfile "$HOST_FILE" -n "$nodes" build/$VERSION
        --n "$n" --C0 "$C0_FIRST$C0_SECOND" --C1 "$C1_FIRST$C1_SECOND"
        END_TIME=$(date +%s)

        # Calculate the execution time
        EXEC_TIME=$((END_TIME - START_TIME))
        echo "Execution time for problem size $problem_size, version $VERSION: $EXEC_TIME seconds."

        # Append the execution time to the EXEC_TIMES string
        EXEC_TIMES="$EXEC_TIMES, $EXEC_TIME"
    done

    # Log the result into weak_scaling.txt
    echo "$problem_size, $nodes$EXEC_TIMES" >> versions_weak_scaling.txt

    # Clean up the downloaded file
    rm challenge.txt

    # Double the number of nodes for the next iteration
    nodes=$((nodes * 2))
done
```

A.2 run_stress_test.sh

The `run_stress_test.sh` script performs stress tests to evaluate the a specific version under high loads. Its full content is shown below:

```
#!/bin/bash

INPUT_FILE="$OAR_NODE_FILE"
HOST_FILE="mpi_hostfile"

# Deduplicate and count slots for each node
awk '{slots[$1]++} END {for (host in slots) print host " slots=" slots[host]}'
"$INPUT_FILE" > "$HOST_FILE"

OUTPUT_FILE="stress_test_output.txt"
> "$OUTPUT_FILE"  # This ensures the file is empty at the start of each run

# Run the MPI application in the background
mpiexec --hostfile "$HOST_FILE" --verbose -n 8 build/version4 --n 31
--C0 0e3b626cd41b8a62 --C1 e3637e86c789bc99
```