Parallel Optimization of Large-Scale Genomic Processing with OpenMP

1st Garcia Navarro A
Biomedical Physics.
Faculty of Science.
CDMX, Mexico.
arygarcianav@ciencias.unam.mx

2nd Hernández Pérez F.
Biomedical Physics.
Faculty of Science.
CDMX, Mexico.
ferch$_2$020@$ciencias.unam.mx$

3rd Martínez Cruz A.
Biomedical Physics.
Faculty of Science
CDMX, Mexico,
aldomc@ciencias.unam.mx

4th Tenorio Camacho J.
Biomedical Physics.
Faculty of Science.
CDMX, Mexico.
javiertenorio@ciencias.unam.mx

***Abstract*—The rapid increase of genomic data poses significant challenges for efficient analysis of large tables. In this work, sequential and parallel versions (C/C++ with OpenMP) of two critical operations are implemented and evaluated: column-wise standardization using Z-Score and threshold search. The case study employs the dataset EBPlusPlusAdjustPANCAN_IlluminaHiSeq_RNASeqV2.geneExp.tsv from the TCGA Pan-Cancer Atlas, which contains approximately 2.27×10^8 values. Detailed measurements of execution time, speedup, and efficiency under different thread configurations and scheduling policies are presented. The results show substantial improvements over the sequential version and validate the correctness of the parallel operations. Finally, limitations related to memory bandwidth are discussed, and additional optimizations are proposed.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

The increasing availability of genomic data has driven the need to develop computational tools capable of processing large volumes of information within reasonable time frames. Despite continuous advances in hardware, parallel programming remains an essential approach for efficiently leveraging available resources and reducing overall execution time [**?**].

OpenMP is a widely used API for implementing shared-memory and multithreaded parallelism, offering a flexible and portable structure under the *fork-join* model [**?**]. This framework facilitates the development of parallel programs with minimal modification to the sequential code.

The process is based on a direct task decomposition that can be executed in parallel and then assigned to the different available cores or processors, with each processor executing its assigned task. Finally, the system coordinates and gathers the resulting data and combines them to produce a final output [**?**].

OpenMP implements parallelism using a fork-join model:

- Sequential region: initially, the program runs sequentially with a single thread called the master thread.
- Fork: when the master thread encounters a directive, it spawns a set of slave threads.
- Parallel region: the master and slave threads execute the instructions of the parallel region simultaneously (this is where parallel execution starts).
- Join: at the end, slave threads join the master to continue program execution.

One of the main advantages of using OpenMP is that models created are highly portable and compatible with platforms that support the standard [**?**].

The use of parallel computing for genomic data analysis usually focuses on the mapping or alignment of reads. This process consists of taking millions of short DNA fragments generated by the sequencer and aligning them against a reference genome. Once the reads are aligned, the next step is to identify differences between the sample DNA and the reference genome.

Projects such as [**?**] have used OpenMP to implement data-level parallelism with the main task of aligning a read assigned to multiple CPU threads, achieving significant acceleration by reducing alignment time from days to hours and allowing the program to scale linearly with the number of available cores.

II. Objectives

General objective: Optimize the processing of a gene expression dataset through parallelization with OpenMP and evaluate its performance using high-performance computing metrics.

Specific objectives:

- Implement efficient data structures for large matrices.
- Design sequential and parallel versions of three key algorithms.
- Measure execution times, speedup, and efficiency for different numbers of threads.
- Analyze the scalability of the developed system.

III. Materials and Methods

A. Dataset Description

The dataset used corresponds to the *TCGA Pan-Cancer Atlas*, specifically the file `EBPlusPlusAdjustPANCAN_IlluminaHiSeq_RNASeqV2.geneExp.tsv`. The matrix contains:

- $N =$ 20,531 genes
- $M =$ 11,069 samples
- Approximately 2.27×10^8 floating-point values.

Each row represents a gene and each column a TCGA patient sample.

Identify applicable funding agency here. If none, delete this.

B. Data Structure and Memory Management

Given the size of the dataset, a memory structure allowing fast and continuous access to elements was used. Instead of nested arrays like `std::vector<std::vector<double>>`, which reduce cache efficiency, the entire matrix was stored in a single one-dimensional vector. Each row remains contiguous in memory, and any element can be located via:

$$\text{index} = i \times M + j. \quad (1)$$

Here, i is the row index (gene), j is the column index (sample), and M is the total number of columns, mapping any pair (i, j) to a unique position in the linear vector.

This improves performance during matrix traversal because linear access favors spatial locality and reduces cache misses during statistical or columnar transformations. File reading is separated from computation to measure only processing time. A robust numerical conversion function handles missing or malformed entries without halting execution.

C. Algorithms and Processing

The calculations focused on three main operations:

1) Column-wise statistical calculation: For each patient j, the mean μ_j and sample standard deviation σ_j were calculated:

$$\mu_j = \frac{1}{N}\sum_{i=0}^{N-1} x_{ij}, \qquad \sigma_j = \sqrt{\frac{1}{N-1}\sum_{i=0}^{N-1}(x_{ij}-\mu_j)^2}. \quad (2)$$

2) Z-Score transformation and threshold search: Each element was standardized via:

$$z_{ij} = \frac{x_{ij}-\mu_j}{\sigma_j}. \quad (3)$$

Simultaneously, all values exceeding a threshold (Threshold = 1000.0) were recorded, storing gene name, corresponding sample, and original value.

3) Top-K identification of high expression levels: To efficiently identify the most expressed genes in each sample, a *Top-K* procedure was implemented. This retrieves only the highest values without fully sorting each column, which is computationally expensive given 20k+ genes per patient.

For each column j:

$$\text{TopK}(j) = \{\text{the } K \text{ largest values in } \{x_{0j}, \ldots, x_{(N-1)j}\}\}. \quad (4)$$

The implementation extracts column data into a temporary vector and applies `std::partial_sort`. The first K elements are the largest and sorted in descending order. Time complexity is $O(N \log K)$, easily parallelized across patient columns.

D. Parallel Implementation with OpenMP

Two versions were developed: sequential and parallel. Parallelization was applied in:

- **Column statistics stage:** `#pragma omp parallel for schedule(static)` assigns blocks of columns uniformly. Static scheduling reduces overhead, as all columns require similar computation.
- **Z-score, threshold search, and Top-K stage:** Loops were parallelized with `#pragma omp parallel for schedule(dynamic)`. Dynamic scheduling adapts to workload variability, particularly during threshold search. Each thread uses a private vector (`local_hits`) to avoid race conditions. Once finished, results are integrated into a global vector using a `critical` section. Top-K calculation is parallelized per patient.

This ensures correctness and efficient memory use.

E. Performance Metrics

Performance was measured as total computation time in sequential and parallel phases. Standard HPC metrics were calculated:

$$S_p = \frac{T_{seq}}{T_p}, \qquad E_p = \frac{S_p}{p} \times 100\%. \quad (5)$$

Tests varied the number of threads ($p = 4, 8$) to assess scalability and parallel efficiency.

IV. RESULTS

Execution times recorded were:

$$T_1 = 143.21 \text{ s} T_4 = 31.88 \text{ s} T_8 = 27.10 \text{ s}$$

The corresponding speedup and efficiency values are shown in Table I.

TABLE I
SPEEDUP AND EFFICIENCY RESULTS

Threads	Time (s)	Speedup	Efficiency
1	143.21	1.00	1.00
4	31.88	4.49	1.12
8	27.10	5.28	0.66

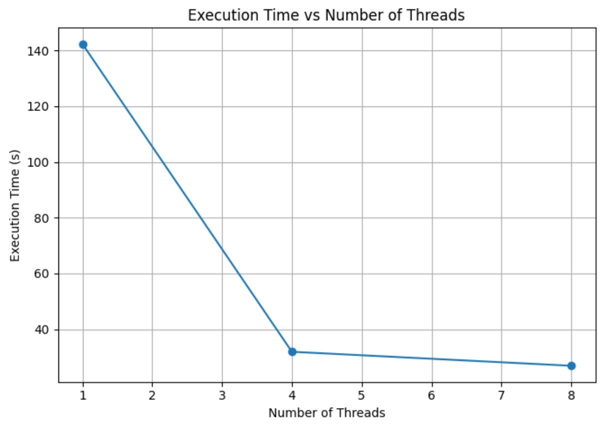

Fig. 1. Execution time vs number of threads.

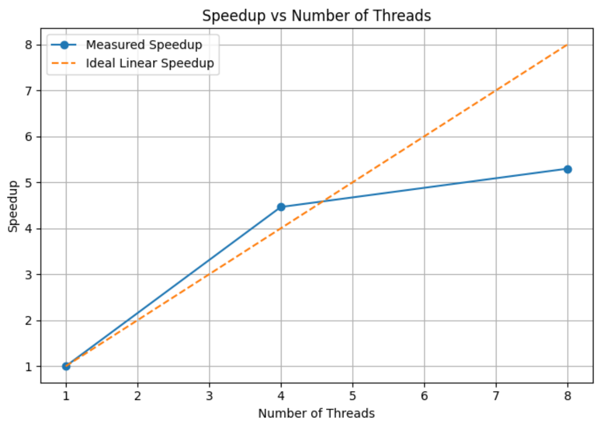

Fig. 2. Actual speedup vs ideal speedup.

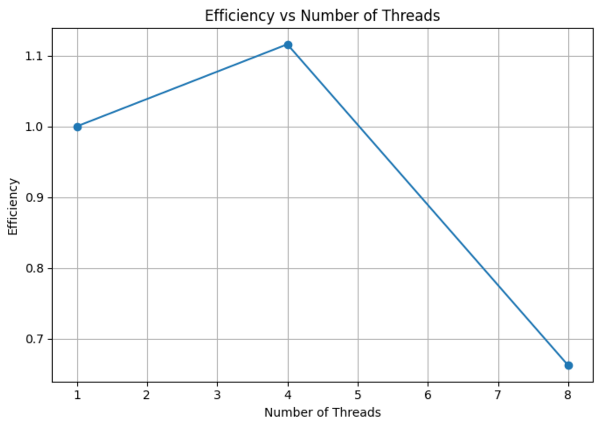

Fig. 3. Efficiency vs number of threads.

V. Discussion

Results show a notable decrease in execution time as the number of threads increases. With 4 threads, execution time is reduced by over 75% compared to the sequential run.

The speedup with 8 threads (5.28) does not reach the ideal value of 8 due to:

- thread creation and management overhead,
- non-parallel algorithm regions (Amdahl's Law),
- early saturation of CPU functional units.

Efficiency over 1 (112%) for 4 threads indicates *superlinear speedup*, often due to:

- better CPU cache usage,
- more effective problem partitioning,
- reduced memory thrashing.

For 8 threads, efficiency drops to 0.66, indicating additional threads no longer yield proportional gains. This is typical when hardware parallelism reaches its limit.

VI. Conclusion

The parallel implementation significantly improves performance as the number of threads increases. However, gains are not linear, and beyond a certain point, adding threads provides no substantial benefit. Results align with theoretical parallelism principles.

VII. CÓDIGO FUENTE

```

#Paralell
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <sstream>
#include <cmath>
#include <chrono>
#include <algorithm>
#include <iomanip>
#include <omp.h>

using namespace std;

//estructuras
struct Hit {
    string gene;
    string patient;
    double value;
};

struct TopKItem {
    double value;
    int geneIndex;
};

struct Dataset {
    int rows = 0;
    int cols = 0;
    vector<string> colNames;
    vector<string> geneNames;
    vector<double> data;
};

// funcion para cargar el archivo
double safe_stod(string s) {
    if (s.empty()) return 0.0;
    if (s.back() == '\r') s.pop_back();
    if (s ==  NA  || s ==  na  || s ==  null  || s ==  NaN ) return 0.0;
    try { return stod(s); } catch (...) { return 0.0; }
}

Dataset loadCSV(const string& filename, int maxLines) {
    Dataset db;
    ifstream file(filename);
    if (!file.is_open()) {
        cerr <<  Error FATAL: No se pudo abrir   << filename << endl;
        exit(1);
    }
    string line;
    if (getline(file, line)) {
        if (!line.empty() && line.back() == '\r') line.pop_back();
        stringstream ss(line);
        string cell;
        getline(ss, cell, '\t');
        while (getline(ss, cell, '\t')) db.colNames.push_back(cell);
    }
    db.cols = db.colNames.size();

    cout <<  Cargando datos a RAM...  << endl;
    while (getline(file, line) && (maxLines == -1 || db.rows < maxLines)) {
        if (line.empty()) continue;
        size_t tabPos = line.find('\t');
        if (tabPos == string::npos) continue;
        db.geneNames.push_back(line.substr(0, tabPos));
        size_t start = tabPos + 1;
        size_t end;
        while ((end = line.find('\t', start)) != string::npos) {
            db.data.push_back(safe_stod(line.substr(start, end - start)));
            start = end + 1;
        }
        db.data.push_back(safe_stod(line.substr(start)));
        db.rows++;
    }
    file.close();
    return db;
}

//procesamiento paralelo
void process_parallel(Dataset& db, double threshold, vector<Hit>& global_hits, vector<vector<TopKItem>>& all_topk, int num_threads) {

    omp_set_num_threads(num_threads);

    vector<double> means(db.cols, 0.0);
    vector<double> std_devs(db.cols, 0.0);
    const int K = 10;

    #pragma omp parallel
    {
        #pragma omp for schedule(static)
        for (int j = 0; j < db.cols; j++) {
            double sum = 0.0;
            for (int i = 0; i < db.rows; i++) sum += db.data[i * db.cols + j];
            means[j] = sum / db.rows;

            double sum_sq = 0.0;
            for (int i = 0; i < db.rows; i++) {
                double val = db.data[i * db.cols + j];
                sum_sq += pow(val - means[j], 2);
            }
            double variance = (db.rows > 1) ? (sum_sq / (db.rows - 1)) : 0.0;
            std_devs[j] = sqrt(variance);
            if (std_devs[j] == 0.0) std_devs[j] = 1.0;
        }

        vector<Hit> local_hits;

        #pragma omp for schedule(dynamic)
        for (int i = 0; i < db.rows; i++) {
            for (int j = 0; j < db.cols; j++) {
                int idx = i * db.cols + j;
                double val_original = db.data[idx];

                if (val_original > threshold) {
                    local_hits.push_back({db.geneNames[i], db.colNames[j], val_original});
                }
                db.data[idx] = (val_original - means[j]) / std_devs[j];
            }
        }

        #pragma omp critical
        global_hits.insert(global_hits.end(), local_hits.begin(), local_hits.end());

        #pragma omp barrier

```

```cpp
        #pragma omp for schedule(dynamic)
        for (int j = 0; j < db.cols; j++) {
            vector<TopKItem> col_vals;
            col_vals.reserve(db.rows);
            for (int i = 0; i < db.rows; i++) {
                col_vals.push_back({db.data[i * db.cols + j], i});
            }

            std::partial_sort(col_vals.begin(),
                              col_vals.begin() + K,
                              col_vals.end(),
                              [](const TopKItem& a, const TopKItem& b) {
                                  return a.value > b.value;
                              });

            col_vals.resize(K);
            all_topk[j] = col_vals;
        }
    }
}

// MAIN
int main(int argc, char* argv[]) {
    int n_threads = 4;
    if (argc > 1) n_threads = atoi(argv[1]);

    string filename =  data.tsv ;

    int maxLines = -1;
    double umbral = 1000.0;

    Dataset db = loadCSV(filename, maxLines);
    vector<Hit> hits;
    vector<vector<TopKItem>> all_topk(db.cols);

    auto start = chrono::high_resolution_clock::now();
    process_parallel(db, umbral, hits, all_topk, n_threads);
    auto end = chrono::high_resolution_clock::now();

    cout <<  Finalizado exitosamente.  << endl;
    return 0;
}
```