Algoritmo memético paralelo: Avance Final

Pablo Zapata

CSP: Closest string problem

- Encontrar el centro geométrico de un set de strings.
- En palabras más simples, encontrar una string donde la distancia hamming entre ella y cada una de un conjunto sea de a lo más x. Objetivo: minimizar x.
- NP-Hard.
- Útil para encontrar señales en secuencias de ADN.

CSP: Closest string problem

Instancia: C = {s_1, s_2, ..., s_n}, n strings de tamaño m sobre un alfabeto A.

Objetivo: Encontrar una string x de tamaño m sobre A tal que $d_H(x, s_i) \leq d_c$ para todo s_i en C, donde d_c es mínimo.

C = {ACGT, TTAC, CCGC}, x = TCGC

$d_H(x, s_1) = 2$ $d_H(x, s_2) = 2$ $d_H(x, s_3) = 1$

Avances

Implementación de hebras “persistentes” con OpenMP

```
# pragma omp parallel num_threads(hilos) if(hilos)
{
    while(tActual < tiempoMaximo){
        // Crear antibiotico
        # pragma omp single
            crearAntibiotico();

        // Mutacion y busqueda local
        # pragma omp for
            for(Bacteria &b: bacterias){
                if(prob(pm)) b.mutar();
                if(prob(bl)) b.busquedaLocal();
                b.actualizarFitness();
            }

        // Evaluacion y clasificacion de bacterias
        # pragma omp single
        {
            evaluacion();
            clasificacion();
            if(!donadoras.empty()) bacteriaFeromonas = &bacterias[donadoras[rng()%donadoras.size()]];
            else bacteriaFeromonas = &bacterias[rng()%poblacion];
        }

        // Sincronizacion
        # pragma omp barrier
```

Avances

Implementación de hebras “persistentes” con OpenMP

```
        // Actualizar feromonas
        # pragma omp for
            for(int i=0; i<dataset->m; i++){
                char base = bacteriaFeromonas->solucion[i];
                int aux = dataset->feromonas[i][base];
                dataset->feromonas[i][base] = aux*(1.0-rho) + (dataset->m - bacteriaFeromonas->fitness);
            }

        // Crear descendencia
        if(donadoras.size() > 1)
            # pragma omp for
                for(int i=0; i<receptoras.size(); i++){
                    int r1 = rng()%donadoras.size();
                    int r2 = (donadoras[r1] + 1 + rng()%(donadoras.size()-1))%donadoras.size();

                    bacterias[receptoras[i]].hijo(&bacterias[donadoras[r1]], &bacterias[donadoras[r2]]);
                }

        // Calcular tiempo
        # pragma omp single
            tActual = duration_cast<milliseconds>(high_resolution_clock::now() -  ti).count()/1000.0;

        # pragma omp barrier // Sincronizacion
    }
}
```

Avances

Integracion del avance 2

Cada individuo de la población con su propia clase DatasetCUDA

```
__global__ void kernelHamming(char *s1, char *s2, int *dist){
    extern __shared__ int temp[];
    int offset = blockIdx.x*blockDim.x;
    thread_group g = this_thread_block();

    int tid = g.thread_rank();

    if(s1[tid + offset] != s2[tid + offset]) temp[tid] = 1;
    else temp[tid] = 0;
    g.sync();

    for(int i=512; i>0; i/=2){
        if(tid < i && i < blockDim.x) temp[tid] += temp[tid + i];
        g.sync();
    }

    if(threadIdx.x == 0) atomicAdd(dist, temp[0]);
}
```

```
TAATCGT ... GCGGCCC
TCTTCGG ... ACCCATA
0110001 ... 1011111
```

Reducción

Avances

```
DatasetCUDA(int n2, int m2, vector<string> dataset, int identificador){
    n = n2;
    m = m2;
    id = identificador;
    cudaSetDevice(id);

    datasetDst = vector<char*>(n);
    dist = vector<int*>(n);


    cudaMalloc(&solDst, m*sizeof(char));
    for(int i=0; i<n; i++){
        cudaMalloc(&datasetDst[i], m*sizeof(char));
        cudaMallocHost(&dist[i], sizeof(int));
    }

    stream = vector<cudaStream_t>(n);
    for(int i=0; i<n; i++) cudaStreamCreate(&stream[i]);

    bloques = m/1000;
    hebras = 1000;
    sharedBytes = hebras * sizeof(int);

    for(int i=0; i<n; i++)
        cudaMemcpyAsync(datasetDst[i], &dataset[i][0], m*sizeof(char), cudaMemcpyHostToDevice, stream[i]);
}
```

```
class DatasetCUDA{
    private:
        char *solDst;
        vector<int*> dist;
        int bloques, hebras, sharedBytes, n, m, id;
        vector<char*> datasetDst;
        vector<DatasetCUDA> datasetCUDA;
        vector<cudaStream_t> stream;
```

Avances

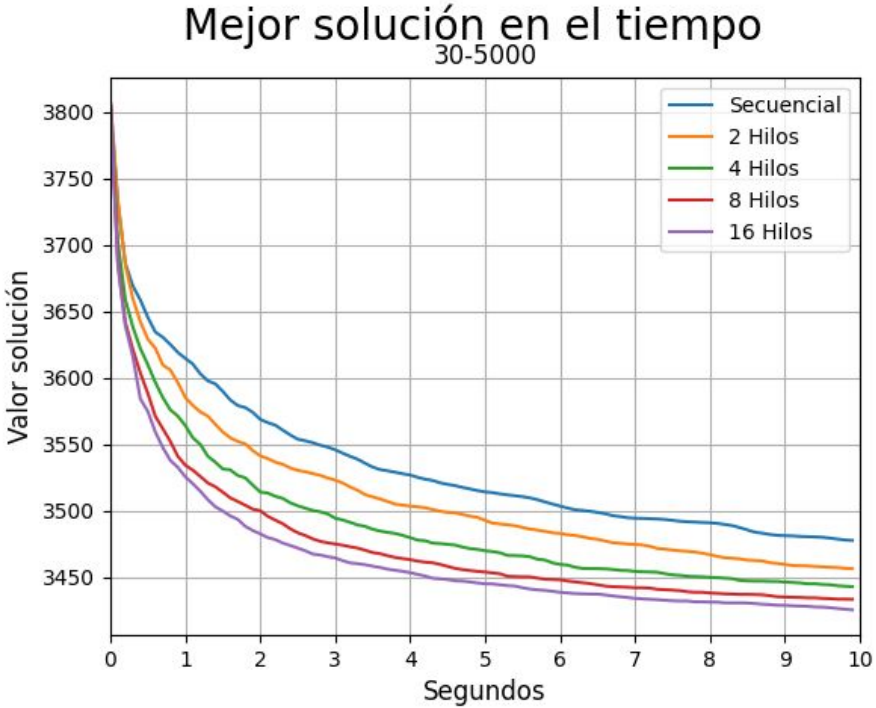
```
vector<int> calidadCUDA(string sol){
    cudaSetDevice(id);
    cudaMemcpyAsync(solDst, &sol[0], m*sizeof(char), cudaMemcpyHostToDevice);
    for(int i=0; i<n; i++) *dist[i] = 0;

    for(int i=0; i<n; i++) kernelHamming<<<bloques, hebras, sharedBytes, stream[i]>>>(solDst, datasetDst[i], dist[i]);
    cudaDeviceSynchronize();

    vector<int> distHamming(n);
    for(int i=0; i<n; i++) distHamming[i] = *dist[i];

    return distHamming;
}
```

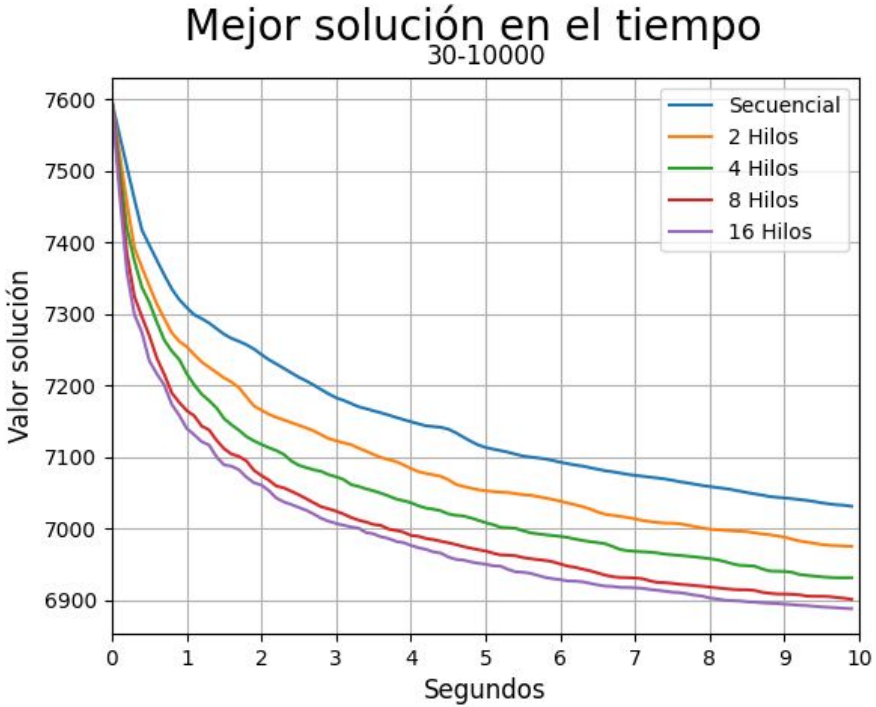
Análisis experimental



n: 30 m: 5000

Poblacion: 50	Tiempo para llegar a solución (Segundos)		
	3600	3550	3480
Secuencial	1.25	2.78	9.78
2 Hilos	0.86	1.8	6.33
4 Hilos	0.57	1.2	3.97
8 Hilos	0.42	0.82	2.78
16 Hilos	0.34	0.68	2.19

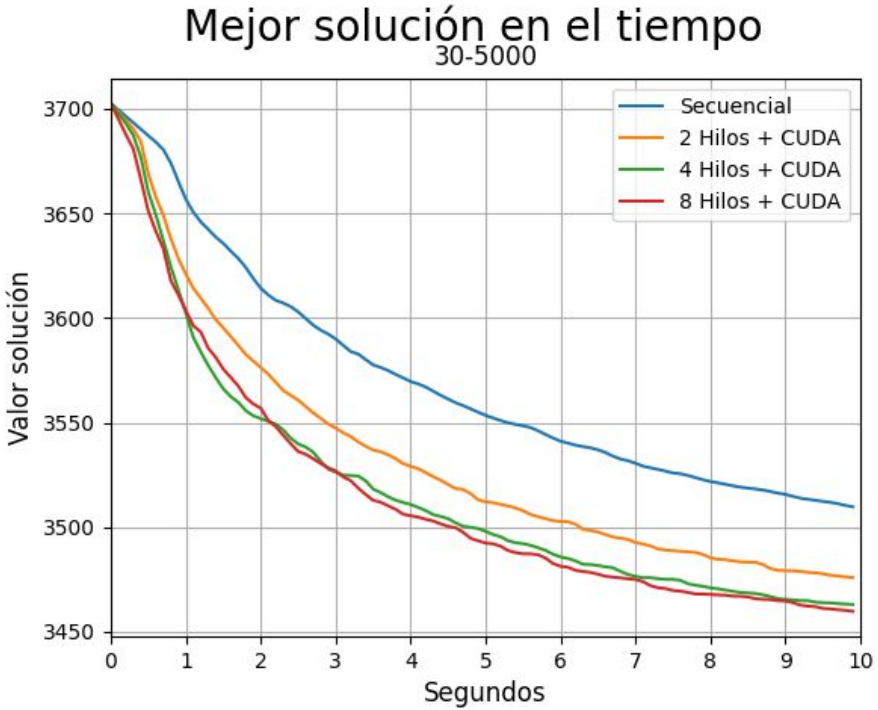
Análisis experimental



n: 30 m: 10000

Poblacion: 50	Tiempo para llegar a solución (Segundos)		
	7200	7100	7040
Secuencial	2.7	5.58	9.2
2 Hilos	1.68	3.58	5.9
4 Hilos	1.01	2.33	3.84
8 Hilos	0.76	1.7	2.62
16 Hilos	0.69	1.39	2.22

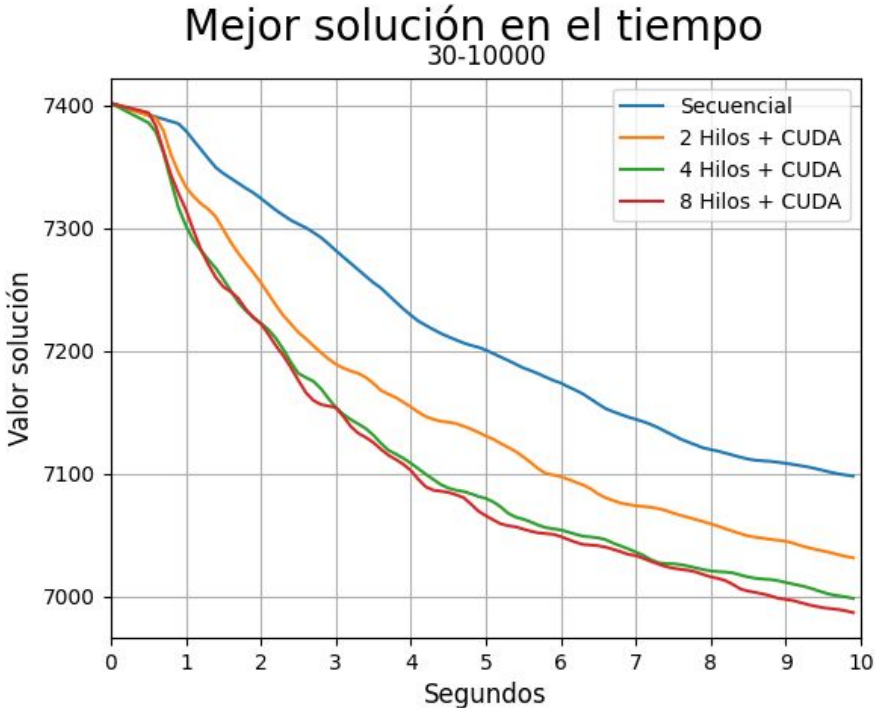
Análisis experimental



n: 30 m: 5000

	Tiempo para llegar a solución (Segundos)		
Poblacion: 50	3600	3550	3510
Secuencial	2.57	5.28	9.82
2 Hilos + CUDA	1.38	2.86	5.29
4 Hilos + CUDA	1.08	2.13	3.85
8 Hilos + CUDA	1.01	2.13	3.69

Análisis experimental



n: 30 m: 10000

Poblacion: 50	Tiempo para llegar a solución (Segundos)		
	7300	7200	7100
Secuencial	2.6	5.02	9.68
2 Hilos + CUDA	1.47	2.76	5.79
4 Hilos + CUDA	1.08	2.25	4.19
8 Hilos + CUDA	0.99	2.24	4.01

Análisis teórico

Calculo distancia hamming CUDA:
O(n + b)

Procesadores: t
Poblacion: p

Pseudocódigo	**Secuencial**	**OpenMP + CUDA**
crearPoblacion()	O(p*n*m)	O(p*n*m)
while(condiciones):		
crearAntibiotico()	O(1)	O(1)
mutacion()	O(p*m)	O(p/t)*(m)
busquedaLocal()	O(p*n*m)	O(p/t)*(n*m)
actualizarFitness()	O(p*n*m)	O(p/t) * O(n + b)
evaluacion()	O(p)	O(p)
clasificacion()	O(p)	O(p)
actualizarFeromonas()	O(m)	O(m/t)
descendencia()	O(p)	O(p/t)
Complejidad del ciclo:	O(p*n*m)	O(p + n*m)