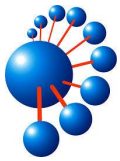


Informe Final
Algoritmo memético paralelo para el CSP

Pablo Zapata Schifferli

Indice

CSP: Closest String Problem

El CSP es un problema de optimización combinatoria. En palabras simples, busca encontrar una string que se parezca lo más posible a un conjunto de strings. Formalmente, sea un conjunto C = $\{s_1, s_2, ..., s_n\}$, con n strings de tamaño m sobre un alfabeto A. Se quiere 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 y $d_h(x, s_i)$ es la distancia hamming entre x y s_i.

El CSP tiene aplicaciones en biología computacional, tales como el descubrimiento de objetivos farmacológicos potenciales, la creación de sondas de diagnóstico, cebadores universales o secuencias de consenso imparciales. Una aplicación común es el diseño de una nueva secuencia de ADN o proteína que es similar a otra secuencia.

Es NP-Completo, por lo actualmente (Hasta que se demuestre si P = NP o no) solo existen algoritmos exponenciales para resolverlo de manera exacta, tales como CPLEX. Debido a esto se han propuesto heurísticas, metaheurísticas, algoritmos genéticos para dar soluciones buenas en un tiempo razonable. Algunas de estas heurísticas son n-aproximado.

Objetivos del proyecto semestral. Desarrollo. Evaluación con presentación de resultados y análisis, esta sección debe incluir descripción de los datos y medidas o métricas utilizadas en la evaluación, Conclusiones

Objetivos del proyecto

Algoritmo memético

Un algoritmo memético es un algoritmo genético combinado con búsqueda local. El algoritmo a usar en este proyecto es el siguiente:

CrearPoblación(): Se crean los individuos con soluciones aleatorias, con una pequeña probabilidad de usar el algoritmo del paper.

while(haya tiempo **):**
  crearAntibiotico(): El antibiótico es el mayor fitness de dos individuos al azar.
  mutacion(): Cada individuo puede sufrir cambios en su solución (string x).
  busquedaLocal(): Cada individuo puede ejecutar el algoritmo del paper.
  evaluacion(): Se calcula el valor d_c para cada individuo, este es su fitness.
  clasificacion(): Los individuos son donadores si fitness ≥ antibiótico, receptores en otro caso.
  actualizarFeromonas(): *
  administrarAntibiotico(): Los receptores son reemplazados por descendencia de donadores

* Las feromonas están en una matriz 4xm, donde cada columna c almacena el valor de feromona de cada base en dicha posición c. Se elige a un donador al azar si existe, si no receptor. Luego, usando la solución del individuo elegido se actualizan los valores de feromonas usando su fitness en la matriz mencionada.

Avance 1

1. Mejorar acceso de memoria al dataset en el algoritmo memético. Se recorre en columnas el dataset en algunas zonas.
2. Implementar el algoritmo del paper e integrarla al algoritmo memético.
    - A parallel multistart algorithm for the closest string problem, Fernando C. Gomes, Cláudio N. Meneses, Panos M. Pardalos, Gerardo Valdisio R. Viana
3. Implementar OpenMP en el algoritmo memético.

Avance 2

1. Implementar el cálculo de distancia hamming con CUDA.

Avance Final

1. Implementación de hebras persistentes de OpenMP en el algoritmo memético junto a la integración del Avance 2

Desarrollo

Avance 1

1. No hubo mejoras de rendimiento al mejorar el acceso a memoria. Principalmente porque los accesos al dataset por columna son pocos.
2. Se implementó el algoritmo de búsqueda local del paper con una modificación, la string del conjunto que se elige para parecerse a ella ahora es elegida de manera aleatoria.
3. Debido a la múltiple cantidad de bucles for que hay en el algoritmo memético, se intentó paralelizarlos usando OpenMP, sin embargo, estos están en un bucle while y no tienen mucho tiempo de ejecución, causando en la mayoría de ellos un menor rendimiento debido a la constante creación de hebras.
    a. La zona donde hubo mejora por parte de OpenMP fue en la zona de mutación (cambios aleatorios en la solución de cada individuo), búsqueda local (algoritmo del paper) y actualización de fitness (cálculo de distancias hamming).

Paralelismo con OpenMP

```
# pragma omp parallel for num_threads(hilos) if(hilos)
        for(Bacteria &b: bacterias){
                if(prob(pm)) b.mutar();
                if(prob(bl)) b.busquedaLocal();
                b.actualizarFitness();
        }
```

Heurística del papel

- En azul ocurre la selección la del string s del conjunto c. Como se mencionó, esto se implementó de manera aleatoria debido a que mejoraba los resultados.

Input: instance $\mathscr{S}_c$, current solution s, distance d_c, and parameter N
Output: resulting solution s and distance d_c
for $k \leftarrow 1$ **to** n **do**
  $d_k \leftarrow d_{\mathrm{H}}(s^k, s)$
  $d'_k \leftarrow d_k$
end
$OldMax \leftarrow d_c$
$NoImprov \leftarrow 0$
While $NoImprov \leqslant N$ **do**
  $b \leftarrow i$ such that $d_{\mathrm{H}}(s^i, s) = d_c$ /* break ties randomly */
  for $j \leftarrow 1$ **to** m such that $s^b_j \neq s_j$ **do**
    $max \leftarrow -1$
    for $k \leftarrow 1$ **to** n such that $k \neq b$ **do**
      if $(s_j = s^k_j)$ **and**$(s^b_j \neq s^k_j)$ **then** $d_k \leftarrow d_k + 1$
      else if $(s_j \neq s^k_j)$ **and** $(s^b_j = s^k_j)$ **then** $d_k \leftarrow d_k - 1$
      if $(max < d_k)$ **then** $max \leftarrow d_k$
    end
    if $d_c \geqslant max$ /* this is not worse */ **then**
      $d_c \leftarrow max$; $s_j \leftarrow s^b_j$
      for $k \leftarrow 1$ **to** n **do** $d'_k \leftarrow d_k$
    else
      for $k \leftarrow 1$ **to** n **do** $d_k \leftarrow d'_k$
    end
  end
  if $OldMax \neq d_c$ **then**
    $OldMax \leftarrow d_c$
    $NoImprove \leftarrow 0$
  else
    $NoImprov \leftarrow NoImprov + 1$
  end
end

Avance 2

Se implementó el cálculo de distancia hamming usando CUDA. Este consiste en la siguiente idea:

- Se calcula la distancia hamming de una string x con cada string s del conjunto de manera paralela usando streams y se guardan los resultados en un arreglo.
- Cada hilo del kernel, con un identificador tid, se encarga de comparar un solo carácter de la string x y de la string s del conjunto que recibió, x[tid] vs s[tid]. Luego si hay diferencia o no escribe un 1 o 0 en la memoria compartida respectivamente, temp[tid] = valor. Finalmente se hace una reducción de temp para obtener la distancia hamming entre x y s.

Las estructuras de datos están dentro de una clase al igual que las funciones, a excepción del kernel, procesarCUDA() se ejecuta una sola vez al inicio del algoritmo, debido a que solo se necesita cargar el conjunto de strings c (dataset) en la gpu una sola vez, lo mismo con los streams, entre otros.

```
vector<string> dataset;
char *solDst;
vector<char*> datasetDst;
vector<int*> dist;
vector<cudaStream_t> stream;
int bloques, hebras, sharedBytes;

void procesarCUDA(){
	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]);
}
```

Función para calcular las distancias hamming de una string solución con el dataset.

```
vector<int> calidadCUDA(string sol){
	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;
}
```

Kernel

```
__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]);
}
```

Avance Final

Debido al problema de estar creando hebras cada vez que el ciclo while se ejecuta, ahora se implementó de manera que las hebras se crearán antes del ciclo, eliminando el tiempo de creación de hebras y pudiendo hacer uso de estas en otras zonas. También se integró el avance 2, sin embargo, debido a la concurrencia de uso de las estructuras de datos por parte de las hebras de OpenMP, cada individuo de la población recibe sus propias estructuras junto a un identificador para hacer uso de cudaSetDevice().

Implementación de OpenMP

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

      // Mutación, búsqueda local y cálculo de fitness
      # pragma omp for
         for(int i=0; i<poblacion; i++){
            if(prob(pm)) bacterias[i].mutar();
            if(prob(bl)) bacterias[i].busquedaLocal();
            bacterias[i].actualizarFitness();
         }

      # pragma omp barrier

      // evaluación, clasificación y elección de bacteria para actualización de feromonas.
      # pragma omp single
      {
         evaluacion();
         clasificacion();
         bacteriaFeromonas ← elección aleatoria de individuo en donadoras,
                             receptoras si no hay donadoras
      }

      # pragma omp barrier

      // Actualizar feromonas
      # pragma omp for
         for(int i=0; i<m; i++) actualizar feromonas usando bacteriaFeromonas
```

```
        // Crear descendencia
        if(donadoras.size() > 1)
            # pragma omp for
                for(int i=0; i<receptoras.size(); i++) reemplazar a receptora[i] con descendencia de
                                                        donadoras

        // Calcular tiempo
        # pragma omp single
            tActual ← tiempo transcurrido desde el inicio del algoritmo

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

Cambios en la implementación del avance 2

El kernel permanece igual, sin embargo las estructuras de datos y funciones se mueven a una clase independiente:

```
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;

    public:
        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]);
    }

    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;
    }
};
```

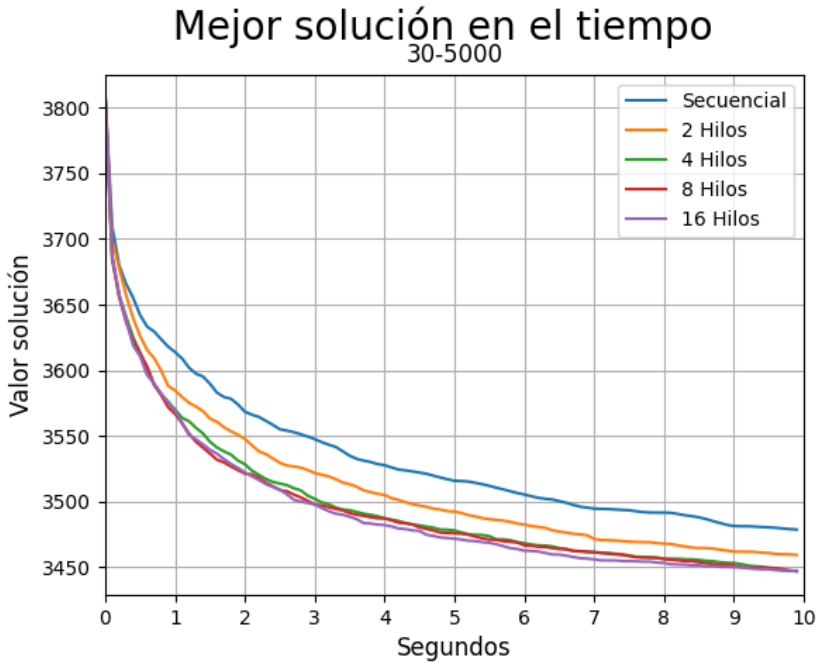
Análisis experimental

Mientras menor sea el valor de la solución mejor.

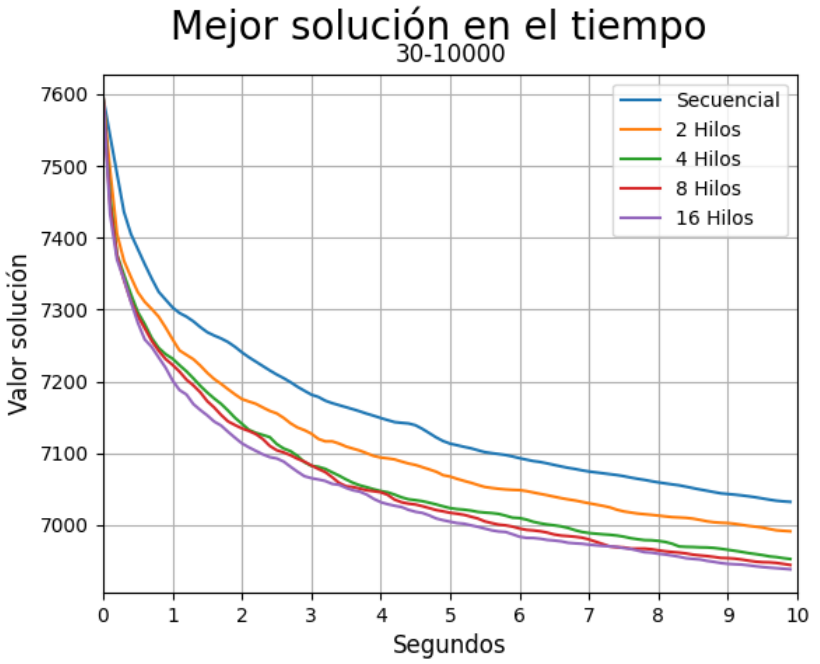
Avance 1

Análisis experimental usando OpenMP en chome. Se puede apreciar que usar más de 4 hilos no ayuda a disminuir el tiempo por mucho, esto debido a la constante creación de hebras por cada ciclo del while.

	Tiempo para llegar a solución (Segundos)		
	3600	3550	3480
Secuencial	1.25	2.78	9.78
2 Hilos	0.79	1.92	6.3
4 Hilos	0.62	1.43	4.69
8 Hilos	0.61	1.22	4.53
16 Hilos	0.57	1.22	4.16



	Tiempo para llegar a solución (Segundos)		
	7200	7100	7040
Secuencial	2.7	5.58	9.2
2 Hilos	1.65	3.78	6.45
4 Hilos	1.33	2.74	4.27
8 Hilos	1.22	2.62	4.13
16 Hilos	1	2.26	3.83



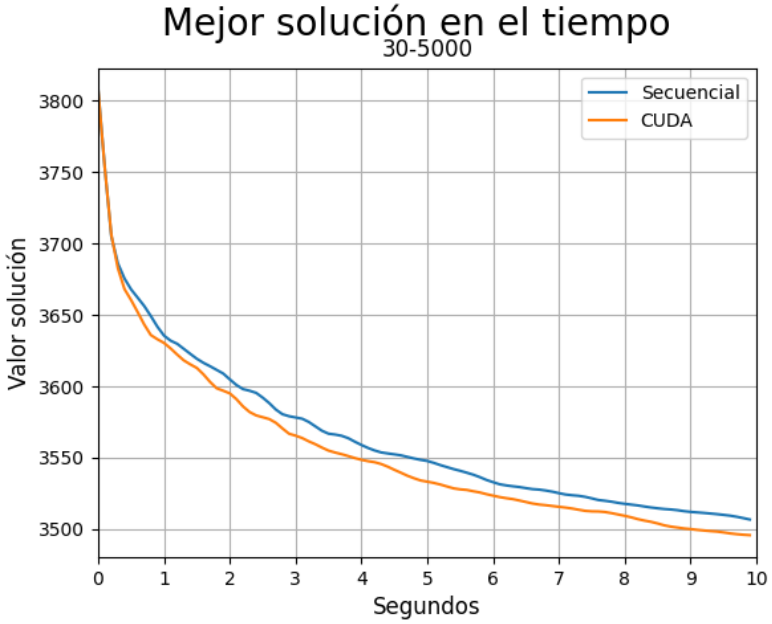
Avance 2

Análisis de tiempo de cálculo de distancia hamming en CUDA vs secuencial

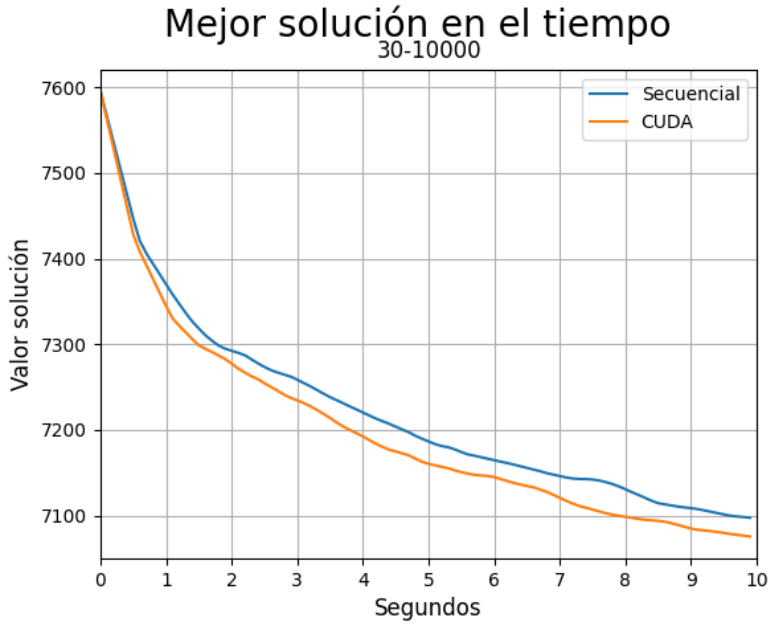
	Milisegundos			Milisegundos	
	Secuencial	CUDA		Secuencial	CUDA
10-1000	0.161	0.087	30-1000	0.492	0.218
10-2000	0.323	0.083	30-2000	0.971	0.219
10-3000	0.487	0.084	30-3000	1.462	0.22
10-4000	0.645	0.085	30-4000	1.938	0.206
10-5000	0.809	0.083	30-5000	2.427	0.205

Análisis experimental del algoritmo memético: Si bien el análisis anterior parecía indicar que podrían haber grandes mejoras, en práctica no las hay, esto debido a que el algoritmo memético pasa la mayor parte de su tiempo ejecutando otras operaciones.

Calidad	Tiempo (Segundos)	
	Secuencial	CUDA
3600	1.91	1.53
3550	4.58	3.78
3510	9.13	7.53



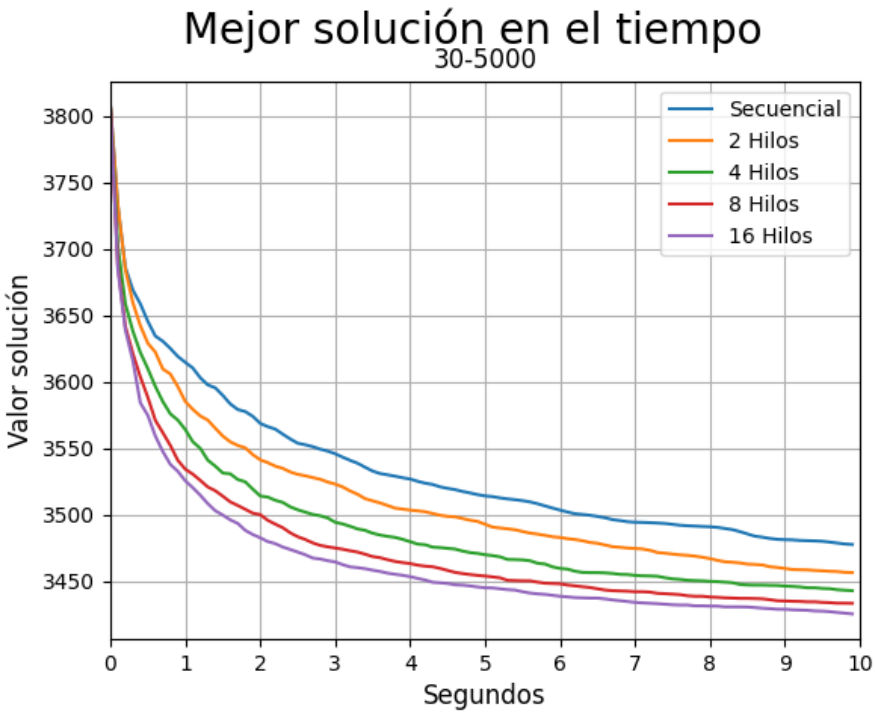
Calidad	Tiempo (Segundos)	
	Secuencial	CUDA
7300	1.77	1.48
7200	4.61	3.8
7100	9.6	7.88



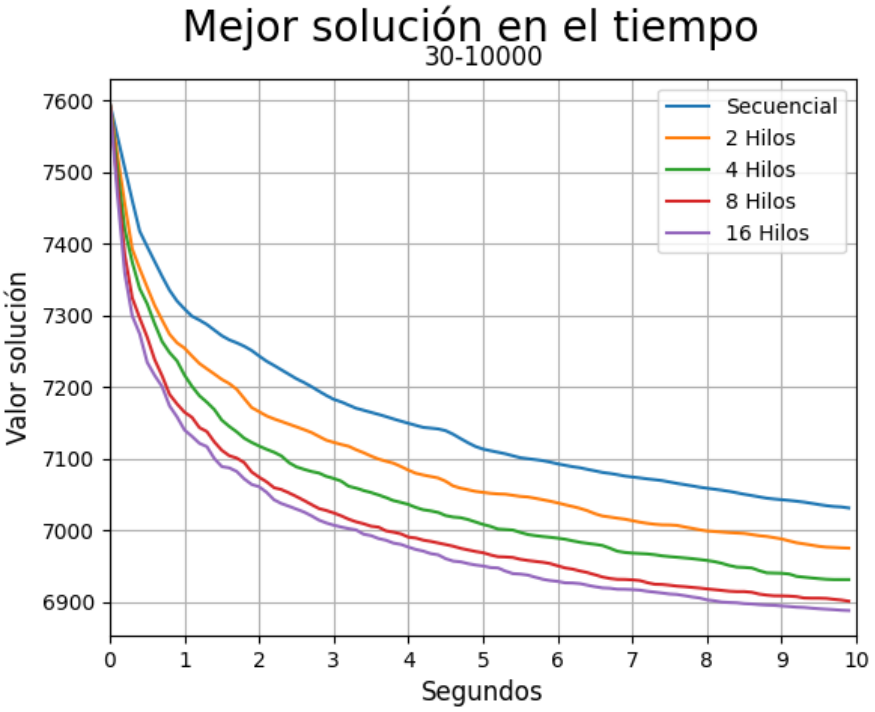
Avance Final

Análisis experimental usando solo OpenMP en chome. Se pueden observar mejores resultados y claras diferencias de rendimiento al agregar más hilos a diferencia del avance 1, esto debido a que las hebras solo se crean una sola vez.

	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

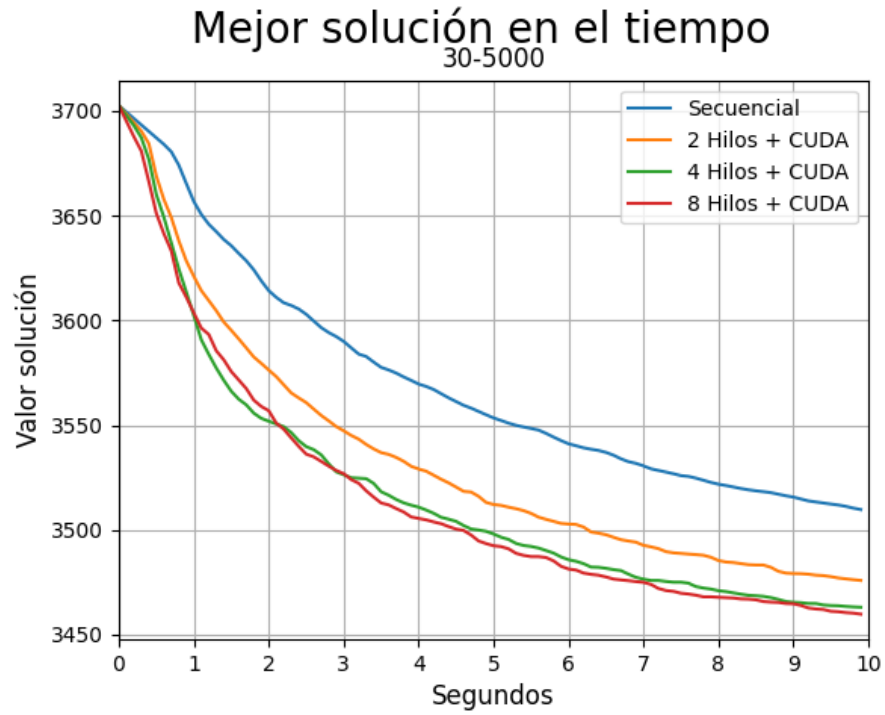


	Tiempo para llegar a solución (Segundos)		
	7200	7100	7040
Secuencial	2.7	5.58	9.2
2 Hilos	1.68	3.58	5.9
4 Hilos	1.01	2.33	3.84
8 Hilos	0.76	1.7	2.62
16 Hilos	0.69	1.39	2.22

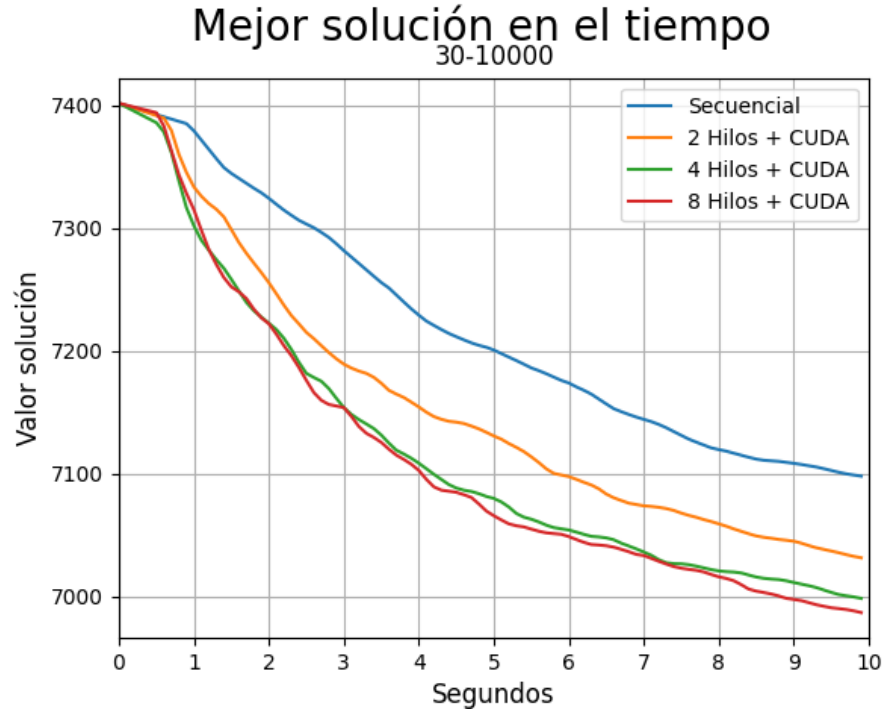


Análisis experimental usando OpenMP junto a CUDA en chome en mi computador. Debido a que cada individuo tiene una copia de las estructuras relacionadas a CUDA, tiene un tiempo de cómputo extra al inicio del algoritmo que se puede apreciar en casi una línea plana. Probablemente el experimento con 8 hilos sea invalido debido a que mi computador tiene 6 núcleos físicos con 12 hilos. Aun así, tiene peor rendimiento que usar solo OpenMP, esto debido por la copia anteriormente mencionada y sumado a que CUDA no aporta mucho rendimiento en este algoritmo como se ve en el avance 2.

	Tiempo para llegar a solución (Segundos)		
	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



	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

Avance 1

El uso de OpenMP ayuda a reducir la complejidad del ciclo si se elige un t en función de p, ejemplo t = p/2.

t: procesadores	Secuencial	Paralelo
while(condiciones):		
crearAntibiotico()	O(1)	O(1)
mutacion()	O(pm)	O(p/t) * O(m)
busquedaLocal()	O(pnm)	O(p/t) * O(nm)
actualizarFitness()	O(pnm)	O(p/t) * O(nm)
evaluacion()	O(p)	O(p)
clasificacion()	O(p)	O(p)
actualizarFeromonas()	O(m)	O(m)
descendencia()	O(p)	O(p)
Total:	O(pnm)	O(p + nm) con t en función de p

Avance 2

La complejidad del cálculo de distancias hamming usando cuda es O(n + b), con b los bloques. Como se mencionó anteriormente, el efecto de usar CUDA no es mucho, esto debido a que el algoritmo pasa más tiempo ejecutando búsqueda local. Tampoco ayuda a reducir la complejidad del ciclo.

	Secuencial	Paralelo
while(condiciones):		
crearAntibiotico()	O(1)	O(1)
mutacion()	O(pm)	O(pm)
busquedaLocal()	O(pnm)	O(pnm)
actualizarFitness()	O(pnm)	O(p) + O(n + b)
evaluacion()	O(p)	O(p)
clasificacion()	O(p)	O(p)
actualizarFeromonas()	O(m)	O(m)
descendencia()	O(p)	O(p)
Total:	O(pnm)	O(pnm)

Avance Final

Como se mencionó anteriormente, el uso de CUDA para el cálculo de distancias hamming no ayuda mucho, y como se aprecia abajo, se puede sacar y no afectará a la complejidad total. A su vez, se puede observar el uso extra de las hebras en actualizarFeromonas() y descendencia() a diferencia del avance 1.

t: procesadores	Secuencial	Paralelo
while(condiciones):		
crearAntibiotico()	O(1)	O(1)
mutacion()	O(pm)	O(p/t) * O(m)
busquedaLocal()	O(pnm)	O(p/t) * O(nm)
actualizarFitness()	O(pnm)	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)
Total:	O(pnm)	O(p + nm) con t en función de p

Concluciones

La paralelización del algoritmo memético dio mejoras limitadas, logrando un rendimiento cuatro veces mayor usando 16 hilos como se observa en el avance final. La paralelización usando CUDA no fue de gran ayuda a pesar de que es bastante más rápida en el cálculo de distancias hamming. El uso de OpenMP y CUDA disminuye el rendimiento a diferencia a solo OpenMP. Tal vez el funcionamiento general del algoritmo añade suficiente overhead en los intentos de paralelización que impide que se pueda paralelizar de buena manera, aunque prefiero pensar de que se puede y no descubrí la manera. Para terminar, si se tienen 16 núcleos libres, es mejor usarlos para un rendimiento 4 veces mayor al del rendimiento secuencial que dejarlos sin usar.