Práctica 3

Metodología de la Programación Paralela
Curso 2024/25

# Paralelización Híbrida con MPI + OpenMP Modelado y Auto-Optimización del Algoritmo Paralelo


Miembros del Grupo:
Miguel Saez Martinez miguel.saezm@um.es
Juan Hernández Acosta juan.hernandeza@um.es

# Índice

**Cuestión 2.**

**Implementar un algoritmo híbrido, tomando como referencia las directivas OpenMP y primitivas MPI que arrojaron mejores resultados en las prácticas 1 y 2. Se deberán utilizar, por tanto, las funciones que se considere más adecuadas para obtener un código optimizado y que permita obtener las mejores prestaciones. Ejecutar el código variando el número de procesos y de hilos, y comparar los resultados obtenidos respecto a la versión secuencial utilizando las métricas de rendimiento de la Tabla 1.**

**Además, se deberá analizar la mejora obtenida en las prestaciones, considerando solo tiempo de ejecución y tiempo de ejecución + fitness (ver apartado (1) de Observaciones), respecto al algoritmo secuencial y también respecto a la mejor versión paralela obtenida en las prácticas anteriores usando el mismo sistema computacional para todas las versiones. 3**

**Si se ejecutó sobre máquinas distintas en prácticas anteriores, volver a tomar tiempos y fitness en un único sistema para comparar las distintas implementaciones.**

**De manera opcional, considerar las indicaciones del apartado (2) de Observaciones.**

Cambios principales en el código

En el código, se integra el paralelismo en dos niveles distintos: primero, se utiliza **MPI** para distribuir la población entre varios procesos, y luego, dentro de cada proceso, se emplea **OpenMP** para que varios hilos trabajen de forma cooperativa. La inicialización de la semilla local se realiza teniendo en cuenta tanto el identificador del proceso (rank) como el del hilo (con omp_get_thread_num()), de modo que cada hilo de cada proceso goza de su propio generador pseudoaleatorio.

La población completa solo existe inicialmente en el proceso maestro (rank=0), que la genera y ordena. A continuación, se reparte a través de un envío asíncrono (con MPI_Isend) al resto de procesos; cada uno recibe su parte local mediante MPI_Irecv. En cada proceso, el código paraleliza las operaciones internas con OpenMP, usando diferentes políticas de scheduling (por ejemplo, schedule(dynamic, chunk_size) o schedule(static, chunk_size)) para equilibrar la carga dependiendo de la naturaleza de cada bucle. Así, la generación inicial de individuos, el cruce de padres y la mutación de genes pueden ejecutarse en paralelo, dividiendo el trabajo entre hilos.

Cada proceso mantiene su subpoblación, evoluciona localmente durante un número determinado de iteraciones (ngm) y, cada cierto número de generaciones (o en la última), se produce una etapa de recolección o migración. En ese momento, cada proceso envía de vuelta su parte de la población al maestro, que la combina y la vuelve a ordenar globalmente para determinar el mejor individuo. El maestro conserva así una visión global de la población y puede imprimir la información de la generación actual si está habilitada la opción PRINT.

En conjunto, la lógica principal reparte el trabajo entre procesos MPI, que a su vez reparten sus subpoblaciones entre hilos OpenMP. Este enfoque híbrido se aprovecha de la escalabilidad de MPI para distribuir la carga entre nodos diferentes y, dentro de cada nodo, de la capacidad de varios núcleos o hilos para abarcar en paralelo las fases más costosas (como la evaluación de la función fitness, la mutación o la creación de nuevos individuos). De esta manera, se logra un equilibrio entre la coordinación necesaria para sincronizar la población global y el alto rendimiento derivado de explotar tanto el paralelismo a nivel de procesos como a nivel de hilos.

Análisis de los resultados

Los resultados se encuentran dentro del documento **Resultados_Prácticas_MPP_EficienciaOrdenada** en la hoja **p3ej2_hibrido** o puede acceder directamente usando este enlace:
Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx

A partir de los datos se aprecia que la eficiencia promedio (0,2469) se encuentra muy por debajo de 1, y lo mismo ocurre con la mediana (0,1206). En otras palabras, en la mayoría de los casos el rendimiento por hilo no llega a igualar el comportamiento ideal que se esperaría (eficiencia = 1). Aun así, el speed-up medio supera ligeramente el valor de 1 (1,6097), lo que indica que existen configuraciones o tamaños de problema en las que se logra una aceleración real respecto al código secuencial.

Sin embargo, al considerar la mediana de speed-up (0,9103), queda claro que en numerosos casos el tiempo de ejecución en paralelo no llega a mejorar el secuencial; es decir, más de la mitad de las ejecuciones presentan un speed-up menor o alrededor de 1, probablemente debido a la sobrecarga de comunicación y sincronización tanto en MPI como en OpenMP.

En conjunto, puede concluirse que el enfoque híbrido (MPI+OpenMP) consigue acelerar la ejecución para ciertos parámetros (especialmente con problemas de mayor tamaño), pero la sobrecarga paralela todavía impide que la eficiencia se mantenga cercana a 1. Esto se hace más evidente en configuraciones pequeñas o cuando el balance de carga no resulta óptimo, ocasionando que parte de los hilos o procesos no alcancen su máximo rendimiento y el speed-up no sea tan alto como el número de hilos podría sugerir en un escenario ideal.

**Cuestión 3.**

**Modelo de Tiempo en Memoria Compartida con 1 nivel de paralelismo.**

## 3.1 Obtención de Datos Experimentales

**a) Paraleliza el bucle for correspondiente a la generación inicial de individuos (FOR_INI) que incluye el cálculo del fitness para cada individuo dentro del mismo bucle for.**

```
    // FOR_INI: generar cada individuo y su fitness
#pragma omp parallel for num_threads(n_hilos_ini) schedule(static)
    for (i = 0; i < tam_pob; i++)
    {
        poblacion[i] = (Individuo *)malloc(sizeof(Individuo));
        poblacion[i]->array_int = crear_individuo(n, m);

        // Calcula fitness inicial con n_hilos_fit
        fitness_parallel(d, poblacion[i], n, m, n_hilos_fit);
    }
```

**b) Modifica el código de la Práctica 1 para que el programa reciba dos nuevos parámetros: el número de hilos (N_HILOS_INI) para el bucle FOR_INI y el número de hilos (N_HILOS_FIT) para el bucle for (FOR_FIT) dentro de la función que realiza el cálculo del fitness.**

```
// Nuevo prototipo con los dos parámetros extra:
extern double aplicar_mh(
    const double *,
    int, int, int, int,
    int *,
    int, /* n_hilos_ini */
    int  /* n_hilos_fit */
);

static double mseconds()
{
    struct timeval t;
    gettimeofday(&t, NULL);
    return t.tv_sec * 1000 + t.tv_usec / 1000;
}

int main(int argc, char **argv)
{
    // Esperamos 6 parámetros además del nombre del programa:
    //   n, m, n_gen, tam_pob, n_hilos_ini, n_hilos_fit
    if (argc < 7)
    {
        fprintf(stderr, "Uso:\n");
```

```
        fprintf(stderr, "  %s n m nGen tamPob nHilosIni nHilosFit\n", argv[0]);
        return (EXIT_FAILURE);
    }

    int n = atoi(argv[1]);
    int m = atoi(argv[2]);
    int n_gen = atoi(argv[3]);
    int tam_pob = atoi(argv[4]);
    int n_hilos_ini = atoi(argv[5]);
    int n_hilos_fit = atoi(argv[6]);
```

```
   Cálculo de fitness en paralelo (FOR_FIT)
------------------------------------------------------------------------ */
void fitness_parallel(const double *d, Individuo *ind, int n, int m, int n_hilos_fit)
{
    int *elem = ind->array_int;
    double sum = 0.0;

#pragma omp parallel for num_threads(n_hilos_fit) reduction(+ : sum) schedule(static)
    for (int i = 0; i < m - 1; i++)
    {
        for (int j = i + 1; j < m; j++)
        {
            sum += distancia_ij(d, elem[i], elem[j], n);
        }
    }
    ind->fitness = sum;
}
```

**c) Instrumentaliza el código para que permita medir el tiempo de ejecución del bucle FOR_INI al variar el número de hilos (por ejemplo, desde 1 hasta 20, con incrementos de 2: 1, 2, 4, 6, 8, …). En este caso no hay que paralelizar el bucle FOR_FIT.**

```
#!/bin/bash

make clean
make sec

# Parámetros fijos
n=30000
m=28000
n_gen=0
tam_pob=6

# Bucle externo: n_hilos_ini en {1,2,4,6,8,10,12,14,16,18,20}
for n_hilos_ini in 1 2 4 6 8 10 12 14 16 18 20
do
    # Bucle interno: n_hilos_fit en {1,2,4,8,16}, por ejemplo
    for n_hilos_fit in 1 2 4 8 16
```

```
    do
        echo "Ejecutando con: n=$n m=$m n_gen=$n_gen tam_pob=$tam_pob
n_hilos_ini=$n_hilos_ini n_hilos_fit=$n_hilos_fit"

        make test_sec \
          N=$n \
          M=$m \
          N_GEN=$n_gen \
          T_POB=$tam_pob \
          N_HILOS_INI=$n_hilos_ini \
          N_HILOS_FIT=$n_hilos_fit
    done
done
```

```
    // 1) Instrumentación: medir tiempo de FOR_INI
    double t_ini_inicial = omp_get_wtime(); // <-- Captura tiempo inicial

    // FOR_INI: generar cada individuo y su fitness
#pragma omp parallel for num_threads(n_hilos_ini) schedule(static)
    for (i = 0; i < tam_pob; i++)
    {
        poblacion[i] = (Individuo *)malloc(sizeof(Individuo));
        poblacion[i]->array_int = crear_individuo(n, m);

        // Calcula fitness inicial con n_hilos_fit
        fitness_parallel(d, poblacion[i], n, m, n_hilos_fit);
    }

    double t_fin_inicial = omp_get_wtime(); // <-- Captura tiempo final de FOR_INI
    double tiempo_for_ini = t_fin_inicial - t_ini_inicial;

    // Imprimir el tiempo medido de FOR_INI
    printf("Tiempo FOR_INI con %d hilos: %.6f seg\n", n_hilos_ini, tiempo_for_ini);
```

**d) Aplicar el algoritmo sobre una instancia del problema de tamaño n = 30000 y m = 28000 (u otros valores que consideres oportunos para tu sistema computacional de manera que, para una correcta visualización del gráfico, el tiempo de ejecución con 1 hilo se sitúe en el intervalo de 5 a 10 segundos). Fija el valor del parámetro NE en la ecuación 1, por ejemplo, NE = 6 (igual al número de cores si el sistema es hexacore) y el número de iteraciones del algoritmo a 0 para que sólo se ejecute la parte de generación inicial de la población (para experimentar con la metodología presentada sólo nos interesa esta función, aunque el proceso real se ampliaría a todas las rutinas del algoritmo). Lanza 5 series de ejecuciones y toma la media de los tiempos experimentales obtenidos con cada número de hilos.**

A la vista de estos datos, puede apreciarse que la configuración que asigna veinte hilos para la fase de creación de la población (n_hilos_ini = 20) y un solo hilo para la fase de fitness (n_hilos_fit = 1) genera los menores tiempos de ejecución globales dentro de las combinaciones probadas. Los resultados muestran cómo, al aumentar el número de hilos en la parte dedicada a generar la población (FOR_INI), se aceleran esas rutinas intensivas en

cómputo, mientras que mantener el cálculo del fitness en un solo hilo no penaliza de manera apreciable el tiempo total para este caso (dado que el número de iteraciones del algoritmo se ha fijado en cero). Por tanto, en este escenario concreto, la estrategia que concentra el paralelismo en la fase inicial de la población ofrece los mejores rendimientos y se traduce en los valores de tiempo de ejecución más reducidos que aparecen en la tabla.

3.2 Instalación de la Rutina (bucle for) con 1 Nivel de Paralelismo

**a) Analiza la hoja de cálculo "Modelo 1-Nivel Paralelismo" del documento datos/solver.ods. En ella se presentan tiempos experimentales (t_experimental) frente al número de hilos, h, así como los correspondientes al modelo teórico (t_teorico) dado por la ecuación 1. Obviamente, debes reemplazar estos datos por tus tiempos obtenidos en el Apartado 3.1.**

**b) Instalar la Rutina con 1 Nivel de Paralelismo (obtener valores de constantes del sistema) a partir de los datos experimentales obtenidos en el Apartado 3.1.**

3.3 Ejecución

**a) Despeja el número de hilos óptimo ($\hbar opt$) de la ecuación 3 en función de NE, $ks$1 y $kh$, donde los valores de $ks$1 y $kh$ han sido obtenidos en la instalación (Apartado 3.2)**

**b) Calcula el tiempo de ejecución que se obtendría con varios valores de h (p.ej.: 3, 5, 7, …) distintos a los usados en la obtención del modelo en los Apartados 3.1 y 3.2.**

**c) Realiza las correspondientes ejecuciones para ese número de hilos con NE = 6 (o el valor que hayas elegido para tu sistema) y comprueba la bondad del ajuste del modelo a los datos experimentales.**

**d) Varía el tamaño de la población dando valores a NE (por ejemplo 3, 10 u otros que estimes oportunos) y obtén el número de hilos óptimo ($\hbar opt$) en estos casos. De esta forma, al variar el tamaño del problema no necesitamos volver a determinar experimentalmente el número de hilos óptimo, sino que podemos hacer el cálculo directamente gracias al modelo de tiempos.**

```
#!/bin/bash

# Compilar con make (limpia antes, opcional)
make clean
make sec

# Parámetros fijos
n=30000
m=28000
n_gen=0

# Bucle para tam_pob entre 3 y 10
```

```
for tam_pob in {3..10}
do
    # Bucle anidado para n_hilos_ini
    for n_hilos_ini in 1 2 4 6 8 10 12 14 16 18 20
    do
        # Bucle interno para n_hilos_fit
        for n_hilos_fit in 1 2 4 8 16
        do
            echo "Ejecutando con: n=$n, m=$m, n_gen=$n_gen, tam_pob=$tam_pob,
n_hilos_ini=$n_hilos_ini, n_hilos_fit=$n_hilos_fit"

            make test_sec \
              N=$n \
              M=$m \
              N_GEN=$n_gen \
              T_POB=$tam_pob \
              N_HILOS_INI=$n_hilos_ini \
              N_HILOS_FIT=$n_hilos_fit
        done
    done
done
```

**e) Ejecuta de nuevo la rutina FOR_INI con NE = {3, 10} y compara los resultados experimentales obtenidos con los arrojados por el modelo. ¿Son similares? ¿Ajusta bien el modelo?**

En estos experimentos, el tamaño de la población varió entre 3 y 10, y se observa que los menores tiempos de ejecución global corresponden precisamente a los valores de población más pequeños (en torno a 3 o 4). Esa diferencia indica que, en este entorno concreto, cuando la población crece más allá de cierto umbral, la sobrecarga de coordinar a un mayor número de individuos empieza a neutralizar la ganancia potencial de disponer de muchos hilos. Además, dentro de estos valores bajos de tamaño de población, los resultados muestran que se alcanzan tiempos aún mejores en combinaciones donde el número de hilos en la fase de generación de la población (n_hilos_ini) no es demasiado elevado y se opta por un número reducido de hilos para el cálculo del fitness (n_hilos_fit). Dicho de otra forma, si bien en ocasiones puede resultar ventajoso paralelizar intensamente ambas fases, en este caso concreto la mejor estrategia parece ser limitar tanto el tamaño de la población como la cantidad de hilos empleados en cada una de las rutinas, de modo que la comunicación y la coordinación no se conviertan en un cuello de botella que termine mermando el rendimiento.

**Cuestión 4.**

**Modelo de Tiempo en Memoria Compartida con 2 niveles de paralelismo. En este caso hay que paralelizar los bucles FOR_INI y FOR_FIT. Recuerda hacer uso de la función omp_set_nested(1) que permite habilitar el uso de paralelismo anidado en OpenMP.**

**Calcula el tiempo de ejecución que se esperaría para varios valores de h1 y h2 (distintos a los usados para instalar el modelo) con la configuración de parámetros utilizada en 4.1. Por otra parte, varía el tamaño de la población dando valores diferentes a NE y |B| y obtén el número de hilos óptimo $h1,_{opt}$ y $h2,_{opt}$ para cada configuración (no hay que lanzar ejecuciones).**

En esta tanda de pruebas, se han modificado tanto el valor de m como el tamaño de la población (tam_pob) para ver cómo influye en los tiempos de ejecución. Al observar los resultados, se nota que los valores más bajos de *tiempo_for_ini* y *exe_time* rondan los 0.50 y 0.51 segundos, y suelen aparecer cuando se combinan niveles de paralelismo moderados o altos en la fase de creación de la población (n_hilos_ini), junto con un número de hilos variado para el *fitness* (n_hilos_fit). Por ejemplo, en las filas donde n_hilos_ini toma valores como 4 u 8 y n_hilos_fit asciende a 16, o incluso donde n_hilos_ini = 20 y n_hilos_fit = 16, se tienden a registrar los tiempos más reducidos.

Esto sugiere que, en este rango de m = 12000 y tam_pob = 3, aumentar en exceso el número de hilos en alguna de las fases puede no traducirse en una mejora lineal, pero sí existe una clara tendencia a que configuraciones con un paralelismo intermedio (por ejemplo, entre 4 y 20 hilos en la fase inicial) logran un equilibrio en el que la sobrecarga de coordinación sigue siendo moderada y la parte computacional está bien distribuida. De esta forma, se consigue que el *tiempo_for_ini* y el tiempo total se mantengan en torno a medio segundo, que es la franja de menor duración en la tabla

**Cuestión 5.**

**Modelo de Tiempo en Memoria Distribuida.**

**Calcula el tiempo de ejecución teórico para diferentes valores de p y resuelve la ecuación 10 para encontrar popt en función de las constantes del sistema (puedes usar algún método numérico para resolverla y utilizar para ello el propio Solucionador).**

En estos resultados, se ve claramente cómo el tiempo de ejecución crece a medida que aumentan las generaciones locales (ngm), pero al mismo tiempo se reduce cuando incrementamos el número de procesos (np), al menos hasta cierto punto. Cuando np = 1, el tiempo sube de manera casi lineal conforme se pasa de 1 a 5 generaciones, llegando a más de 35 segundos. Sin embargo, en cuanto se introducen 2, 4 o 6 procesos, el tiempo para un número bajo de generaciones baja de forma considerable (por ejemplo, a unos 8 o 9 segundos para 1 generación). Esto indica que la mayor parte del coste puede repartirse mejor al tener varios procesos evolucionando la población en paralelo.

No obstante, se aprecia que si se incrementa demasiado el número de procesos (como np = 8), la sobrecarga de comunicación y coordinación empieza a notarse en algunos casos, de modo que no siempre se obtienen los mejores tiempos posibles. En varios puntos, las cifras más bajas corresponden a 4 o 6 procesos; por ejemplo, con 1 generación, np = 4 o np = 6 llevan el tiempo por debajo de 9 segundos, mientras que con np = 8 se está sobre los 10 segundos. Aun así, en un número más alto de generaciones, la diferencia se reduce, en parte porque la fracción de tiempo invertida en comunicación es menos determinante frente al coste computacional de evolucionar la población repetidas veces.

En definitiva, cuanto más grande es ngm, más se encarece la parte evolutiva local en cada proceso. Sin embargo, al usar varios procesos, este coste se reparte y el tiempo total no escala tan rápidamente como cuando solo hay un proceso. Al mismo tiempo, un número excesivo de procesos puede incrementar la sobrecarga de comunicación, por lo que es habitual hallar un valor intermedio (en este caso, con 4 o 6 procesos) que ofrezca los mejores tiempos de ejecución globales.