Universidad de Murcia

Facultad de Informática

TÍTULO DE GRADO EN
INGENIERÍA INFORMÁTICA

Metodología de la Programación Paralela

Práctica Final

Heurísitca: *Algoritmo Genético*

Convocatoria de Enero de 2019

CURSO 2018 / 19

FACULTAD DE
INFORMÁTICA

Profesor: Domingo Giménez Cánovas

Alumno: Kyryl Bogach - kyryl.bogachy@um.es - X5126520G

ÍNDICE

0. Introducción.

Se ha implementado una solución del problema mediante la heurística de un *algoritmo genético*. Pero antes de abarcar cómo se ha planteado el problema, ¿qué es un algoritmo genético?

0.1. ¿Qué es un algoritmo genético?

Un **algoritmo genético** se basa en la evolución inherente en la naturaleza, donde generación a generación se consiguen individuos mejores y más fuertes que son los que sobreviven y perpetúan la especie.

Se parte de una población inicial de individuos que se cruzarán entre ellos en cada generación dando lugar a individuos hijos que serán mejores o peores que los padres.

El ser mejor o peor para un individuo significa su valor fitness, que se obtiene evaluando al individuo con la función que queremos optimizar.

Como es natural, en cada generación pueden surgir mutaciones, lo que añade un componente aleatorio al algoritmo evitando que el algoritmo reduzca mucho su radio de búsqueda en el problema.

El número de generaciones indicará la cantidad de pasadas que se hacen de selección, cruce y mutación de la población de partida.

También se indican la probabilidad de cruce entre individuos seleccionados y la mutación de los genes de los individuos hijos.

0.2. El problema abordado

El problema abordado tiene una serie de parámetros de entrada:

- Número de *personas*.
- Número de *grupos*.
- Número de *asignaturas*.
- Las *asignaturas cursadas* por cada alumno

Además, para este algoritmo se necesitan definir unas características claves:

- *Generaciones*: El número de iteraciones que hará nuestro algoritmo para intentar conseguir el mejor fitness posible. *Por defecto: 1000*
- *Tamaño* de la población. *Por defecto: 200*
- *Probabilidad de Cruce*. *Por defecto: 0.9*
- Probabilidad de Mutación. *Por defecto: 0.1*

Nuestra función de fitness viene dada por la elección de la función "*Min-DesMed*", la cual se obtiene de la media de las diferencias máximas de alumnos en los subgrupos por asignatura. Nótese que, en un algoritmo genético, la función fitness es crucial para obtener un buen resultado.

Sabemos de antemano que no es una función de fitness muy buena ya que si tuviéramos como diferencias máximas [2, 2, 2, 2] nos daría fitness 0. De este modo, para esta práctica, usaremos el algoritmo propuesto para el fitness, pero ponderando por la diferencia máxima, es decir, se aplica la misma metodología, pero se suma al final (al fitness del individuo testado) la máxima diferencia obtenida.

En este caso, con esta mejora, tendríamos un fitness 2 y se seguiría el algoritmo en orden de buscar un mejor fitness. Nótese que estamos haciendo una búsqueda por el **fitness mínimo**.

Se puede quitar esta alteración de la función quitando el parámetro definido en la cabecera de las fuentes adjuntas.

```
// Indica si la función fitness pondera con el máximo
#define MEJORA_FITNESS 1
```

El esquema que sigue un algoritmo genético es el siguiente:

1. Inicializar la población
2. Aplicar la función fitness a todo individuo de la población
3. Seleccionar los mejores individuos
4. Cruce de los individuos
5. Mutación de los individuos

Ahora bien, ¿cómo podemos plantear la paralelización de este algoritmo? ¿Es posible?

0.3. Paralelización del problema

Si nos centramos en una granularidad fina podrían intentar paralelizarse las operaciones de selección, cruce y mutación y la de evaluación de toda la población, con el fin de que se consiga un tiempo de ejecución más rápido. Sin embargo, es posible que la gestión de los hilos y la cantidad de mensajes a enviar pueda ser un poco más costosa.

Podría ser más interesante, por otro lado, interesante paralelizar el algoritmo entero y partir de diferentes poblaciones para poder hacer una mayor cantidad de búsquedas en el mismo tiempo, en vez de conseguir mayor velocidad. De este modo, nos centramos en un grano más grueso.

Se tienen dos estructuras básicas:

1. **Población**: conjunto de individuos.
2. **Individuo**: recoge las asignaciones dadas por este individuo y su fitness para dichas asignaciones.

Para la **inicialización** se crean tantos individuos como el tamaño de la población indique y las asignaciones serán tomadas al azar entres los valores de los grupos [1 - ng].

La función **fitness**, aplicada a cada individuo de la población, se basa en buscar las diferencias máximas y luego aplicar a dichas diferencias el algoritmo "*Min-DesMed*" para obtener un fitness.

La **selección** de Individuos escogida ha sido el algoritmo de "***Selección por Torneo***". Otra opción hubiera sido aplicar el algoritmo de "*Selección por Ruleta***"**. Siguiendo la documentación https://en.wikipedia.org/wiki/Tournament_selection, podemos ver que la selección por torneos binaria es la más utilizada. Es decir, se cogen dos Individuos de la población y "*combaten*" para ver cuál es mejor; el vencedor pasa a ser candidato para el próximo cruce.

El **cruce**, la combinación, está altamente relacionado con la *probabilidad de cruce*. En primera instancia la pondremos en un 90%, ya que, normalmente en un algoritmo genético, la probabilidad de cruce debe ser alta. La probabilidad de cruce es la probabilidad de que se produzca un cruce en un apareamiento particular. De este modo, teniendo los mejores genes escogidos por torneo, cogemos al azar dos individuos que, si la probabilidad de cruce es favorable, se cruzaran del siguiente modo: De manera aleatoria, el 50% de los genes del padre y el 50% de los genes de la madre se van tomando en dos nuevos individuos, el hijo y la hija que, posteriormente, pasaran a formar parte de la nueva población.

La **mutación**, la mejora**,** y su directa relación con la *probabilidad de mutación,* sirve para la diversificación, para no estar atascados en soluciones locales. Pondremos la probabilidad, de nuevo según los algoritmos genéticos típicos, al 10%. De este modo, si la probabilidad de mutación es favorable, algunos de los genes del individuo se "*mutarán*" a un valor aleatorio; puede que este *pequeño* sea el que en un futuro tenga la mejor solución global.

Nuestra **condición de fin**, nuestra convergencia, será tomada por finalizada cuando el fitness obtenido sea el mínimo. Cuando tengamos fitness 0 sabemos que tenemos la mejor solución y terminamos de buscar.

Este proceso se repite hasta que se encuentra la condición de fin o cuando se terminen de iterara todas las generaciones.

Procedemos a explicar todas las soluciones propuestas:

1. Solución secuencial.

Se presenta la solución secuencial en el fichero "*secuencial.cpp*". Se corresponde directamente con el esquema presentado en el apartado anterior; se sigue el mismo orden y la misma funcionalidad indicada.

Su esquema y pseudocódigo es el siguiente:

```
function algoritmoGenetico() {
	poblacion := generarPoblaciónInicial()
	para todas (GENERACIONES) {
		nuevosIndividuos := seleccionTorneo(poblacion)
		poblacion := cruce(nuevosIndividuos)
		poblacion := mutacion(poblacion)
		medirFitness(poblacion)

		si (mejorIndividuoFitness(poblacion) = 0)
			finalizar_bucle()
	}

	return mejorIndividuoFitness(poblacion)
}

function main() {
	leerDatos()
	inicializarParametros()
	mejorResultado = algoritmoGenetico()
	imprimirResultados(mejorResultado)
}
```

2. Solución Memoria Compartida.

Se nos presenta la incógnita de cómo paralelizar el algoritmo genético que tenemos.

Dos opciones:

1. Abarcar el problema con un esquema relajado.
2. Abarcar el problema con un esquema iterativo.

Ya que, como veremos en el posterior estudio teórico y práctico, si apuntamos a una solución de grano fino, donde buscamos paralelizar los bucles y apartados del algoritmo por separado, vemos que no nos vale la pena paralelizar dichas secciones. Los órdenes de ejecución/tiempo de estas funciones por separado son pequeñas por ello un acercamiento al problema con una solución de grano grueso.

De este modo, lo que podemos hacer es subdividir el problema de la siguiente manera:

1. Ver cuantos hilos están disponibles en el sistema.
2. Calcular cuantas iteraciones le toca a cada hilo: *número de generaciones divido entre el número de hilos.*
3. Asignar a cada hilo un número de iteraciones
4. Todo hilo tendrá, al final de su procedimiento, una población.
5. Cotejo de todas las poblaciones obtenidas para coger aquella que tiene una mejor solución.

De este modo, como el problema es altamente paralelizable, las acciones de cada hilo son independientes. Puede ser el caso que un hilo haya sacado una solución local y otro la mejor solución global del problema.

Su esquema y pseudocódigo es el siguiente:

```
function openmp() {
	Lista<Poblacion> poblaciones
	int iteracionesPorHilo = GENERACIONES / numeroHilos()
	#pragma omp parallel firstprivate(iteraciones) shared(poblaciones)
	{
		poblacion := generarPoblaciónInicial()
		para todas (iteracionesPorHilo) {
			nuevosIndividuos := seleccionTorneo(poblacion)
			poblacion := cruce(nuevosIndividuos)
			poblacion := mutacion(poblacion)
			medirFitness(poblacion)

			si (mejorIndividuoFitness(poblacion) = 0)
				finalizar_bucle()
		}

		poblaciones.añadir(poblacion)
	}

	mejorPoblacion = cogerMejorPoblacion(poblaciones)
	return mejorIndividuoFitness(mejorPoblacion)
}

function main() {
	leerDatos()
	inicializarParametros()
	mejorResultado = openmp()
	imprimirResultados(mejorResultado)
}
```

3. Solución Paso de Mensaje.

Para abarcar la solución mediante MPI tenemos que pensar qué datos tenemos que comunicar, cuándo y qué procesos. Además, ¿cuántos proceso vamos a utilizar?

En primer lugar, la idea es utilizar el mismo número de procesos que *cores* físicos disponibles (incluyendo los que obtendríamos con *HyperThreading* y *SMT*) para intentar paralelizar lo máximo posible y poder obtener el mejor tiempo posible a la vez de intentar explorar el abanico de soluciones (soluciones locales vs soluciones globales).

Se propone una solución que realice comunicación entre generaciones, más concretamente se ha pensado que cada 25% de las generaciones se realice una comunicación punto a punto entre los procesos indicando las mejores soluciones (individuos) de estos para que puedan ser añadidas a sus poblaciones actuales.

La inclusión de los nuevos miembros recibidos por los procesos vecinos se hará siguiendo la lógica se sustitución por los individuos con peor fitness, es decir, los que mayor fitness tengan (recordemos que estamos minimizando).

Sin embargo, se nos presenta un problema a resolver: ¿Qué ocurre si un proceso termina y los demás aún no? Esos procesos están esperando un mensaje con las soluciones de *todos* los demás procesos. Una sencilla solución sería ver que, si no hemos terminado nuestras generaciones, comunicar los mensajes que nos faltarían. Si nuestra solución o alguna de las soluciones recibidas tiene un fitness 0 paramos las generaciones.

En cuanto a la comunicación tenemos que comunicar:

1. El proceso 0 debe leer los datos de entrada y comunicarlos a todos los demás procesos. Se hará en dos mensajes diferentes, uno que contenga {*np, ng, na*} y otro que contenga las asignaturas cursadas por los alumnos. Se utilizará una comunicación *Broadcast* para que siga una jerarquía en forma de árbol ya que podría haber muchos procesos. Se utilizan dos mensajes separados porque primero tenemos que saber el tamaño a reservar para las asignaturas, ya que depende de *np* y de *na*; posteriormente sabremos cuantos datos esperamos en la próxima comunicación MPI.
2. Comunicación entre generaciones, donde cada proceso envía punto a punto a todos los demás procesos su mejor solución: las asignaciones actuales y su fitness. Cada proceso incluirá los nuevos individuos en su población de la manera que se ha indicado anteriormente.
3. Comunicación final que consiste en que todos los procesos menos el proceso 0 envíe un mensaje a dicho proceso indicando su mejor solución (su fitness) para poder indicar cual ha sido la mejor solución de todas las encontradas. Se realiza una comunicación punto a punto.

Su esquema y pseudocódigo es el siguiente:

```
function comunicar(poblacion, mejor) {
      para todos proceso menos yo {
            enviar(mejor, proceso)
      }

      individuosRecibidos = {}
      para todos proceso menos yo {
            recibir(mejorRecibido, proceso)
            individuosRecibidos.añadir(mejorRecibido)
      }

      // Los incluimos sustituyendo los individuos
      // con mayor fitness (los peores)
      incluirIndividuos(poblacion, individuosRecibidos)
      return (si algún individuo tiene fitness 0)
}

function mpi(generaciones, comunicacionEntreGeneraciones) {
      poblacion := generarPoblaciónInicial()
      para todas (generaciones) {
            si (generacion % comunicacionEntreGeneraciones = 0) {
                  mejor := cogerMejor(poblacion);
                  si (comunicar(poblacion, mejor))
                        // Algún proceso con fitness 0
                        return mejor.fitness;
            }

            nuevosIndividuos := seleccionTorneo(poblacion)
            poblacion := cruce(nuevosIndividuos)
            poblacion := mutacion(poblacion)
            medirFitness(poblacion)

            si (mejorIndividuoFitness(poblacion) = 0)
                  finalizar_bucle()
      }

      mejor := mejorIndividuoFitness(poblacion)

      // Comunicación restante para otros procesos que esperan
      para todas (generaciones) restantes {
            si (generacion % comunicacionEntreGeneraciones = 0) {
                  si (comunicar(poblacion, mejor))
                        // Algún proceso con fitness 0
                        finalizar_bucle()
            }
      }

      return mejor.fitness;
}

function main() {
      inicializarMPI()
      variablesAleatoriasSegunNodo()
      si (nodo = 0) {
            leerDatos()
            broadcastParametros(np, ng, na, asignaturas)
      } sino {
            broadcastParametros(np, ng, na, asignaturas)
      }
```

```
        inicializarParametros()

        // Comunicación entre generaciones: cada 25%
        comunicacionEntreGeneraciones = GENERACIONES / 4

        // Cada proceso tiene un trozo equitativo de generaciones
        generaciones = GENERACIONES / nº procesos;

        miMejorResultado = mpi(generaciones, comunicacionEntreGeneraciones)

        si (nodo != 0) {
              // Procesos [1, n] envian al proceso 0
              enviar(miMejorResultado, 0)
        } sino {
              // Proceso 0 recibe de los procesos [1, n]
              mejorResultadoGlobal := miMejorResultado

              para todos proceso menos yo {
                    recibir(resultadoRecibido, proceso)
                    si es mejor (resultadoRecibido, mejorResultadoGlobal)
                          mejorResultadoGlobal := resultadoRecibido
              }

              imprimirResultados(mejorResultadoGlobal)
        }
        finalizarMPI()
}
```

4. Solución Híbrida.

La idea para una solución híbrida es utilizar la conjunción de la tecnología *MPI* con *OpenMP*.

Para el máximo aprovechamiento de ambas tecnologías, se propone el uso de la mitad de cores en procesos y la otra mitad en hilos. De este modo, para un sistema con 8 núcleos, tendríamos el siguiente esquema:

4 procesos. Cada proceso haría uso de dos hilos. En total se utilizan las 8 cores.

Se puede optar por, de nuevo, comunicación entre generaciones. Sin embargo, se multiplicarían el uso de mensajes porque cada hilo tendría que comunicarse con todos los demás.

Por ello se propone un esquema más sencillo, donde se divide principalmente el problema con MPI y, posteriormente, en cada proceso se harán uso de dos hilos para maximizar el nivel del paralelismo.

Su esquema y pseudocódigo es el siguiente:

```
function mpiopenmp(generaciones) {
	Lista<Poblacion> poblaciones
	int numHilos = 2
	int iteracionesPorHilo = generaciones / numHilos
	#pragma omp parallel firstprivate(iteracionesPorHilo) shared(poblaciones)
	{
		poblacion := generarPoblaciónInicial()
		para todas (iteracionesPorHilo) {
			nuevosIndividuos := seleccionTorneo(poblacion)
			poblacion := cruce(nuevosIndividuos)
			poblacion := mutacion(poblacion)
			medirFitness(poblacion)

			si (mejorIndividuoFitness(poblacion) = 0)
				finalizar_bucle()
		}

		poblaciones.añadir(poblacion)
	}

	mejorPoblacion = cogerMejorPoblacion(poblaciones)
	return mejorIndividuoFitness(mejorPoblacion)
}

function main() {
	inicializarMPI()
	si (nodo = 0) {
		leerDatos()
		broadcastParametros(np, ng, na, asignaturas)
	} sino {
		broadcastParametros(np, ng, na, asignaturas)
	}

	inicializarParametros()

	// Cada proceso tiene un trozo equitativo de generaciones
	generaciones = GENERACIONES / nº procesos;

	miMejorResultado = mpiopenmp(generaciones)

	si (nodo != 0) {
		// Procesos [1, n] envian al proceso 0
		enviar(miMejorResultado, 0)
	} sino {
		mejorResultadoGlobal := miMejorResultado
		// Proceso 0 recibe de los procesos [1, n]
		para todos proceso menos yo {
			recibir(resultadoRecibido, proceso)
			si es mejor (resultadoRecibido, mejorResultadoGlobal)
				mejorResultadoGlobal := resultadoRecibido
		}

		imprimirResultados(mejorResultadoGlobal)
	}
	finalizarMPI()
}
```

5. Análisis de programas.

Se tienen los programas de las versiones implementadas:

1. secuencial.cpp
2. omp.cpp
3. mpi.cpp
4. mpiopenmp.cpp

Todos estos programas leen por entrada estándar todas las líneas documentadas en la práctica.

Se ha hecho gran uso de un "***Makefile***", el cual se encarga de compilar, ejecutar, generar casos de pruebas, hacer *benchmarks*… etc.

Por otro lado, para la generación de casos tenemos:

1. generador.cpp
2. generador_perfecto.cpp

Donde el primer programa nos da un caso totalmente aleatorio siguiendo como parámetros de ejecución:

- Número de personas a generar
- Número grupos a generar
- Número de asignaturas a generar.
- Probabilidad de matriculación en una asignatura por un alumno (para dar aleatoriedad)

Y el segundo programa genera una salida que puede ser repartida equitativamente entre los grupos teniendo, lo que buscamos, un fitness no. Toma como parámetros:

- Número de personas a generar
- Número grupos a generar
- Número de asignaturas a generar.

Nótese que ambos programas esperan parámetros de ejecución, es decir, no leemos de la entrada estándar.

Ambos ficheros nos servirán para el estudio experimental.

Se adjunta además 3 ficheros de pruebas preestablecidos:

- test_perfecto.txt → fitness 0
- test_mediano.txt → con entrada (*300 10 10*)
- test_grande.txt → con entrada (*1000 15 15*)

6. Estudio teórico.

Los procesadores paralelos se usan para acelerar la resolución de problemas de alto coste computacional.

Consideramos el tiempo en función de los parámetros de entrada y los parámetros del algoritmo genético.

$$t(n,p) = t_a(n,p) + t_c(n,p)$$

6.1. Análisis del tiempo de ejecución.

6.1.1. Secuencial.

Se tiene una suma del coste de:

- Inicializar
- Medir Fitness
- Por cada generación:
  - Selección por Torneo
  - Cruce
  - Mutación
  - Medir Fitness
  - Escoger el mejor Individuo

Inicializar: Θ (T.Población × np)

Medir Fitness: Tras hacer el análisis de todo el recorrido vemos que es bastante denso. Tras quitar las constantes multiplicativas y constantes sumatorias nos queda tal que:

Θ (T.Población × ng² × na² × np)

Selección por Torneo: Θ (T.Población)

Cruce: Θ (T.Población)

Mutación: Θ (T.Población)

Escoger el mejor individuo: Θ (T.Población × ln(T.Población)) → *Una ordenación logarítmica dada por #include <algorithm>*

Si sumamos estos órdenes tenemos que para la versión secuencial tenemos un orden de:

$$\Theta\,(TamPoblacion * (np + ng^2 * na^2 * np)$$

$$+$$

$$Generaciones * (TamPoblacion * (ln(TamPoblacion) + ng^2 * na^2 * np)))$$

6.1.2. OpenMP.

Suponiendo el tiempo secuencial como $t(n)$ lo que se intenta es reducir el orden de ejecución en función del número de cores: $\frac{t(n)}{hilos}$

Como vemos, la parte gruesa del algoritmo está en las iteraciones de las generaciones. Es por ello, que, en este caso, se ha optado por paralelizar esta parte mediante un grano grueso.

Todos los hilos harán:

- Inicializar
- Medir Fitness

Y luego, en función de los $cores$ cogeremos como número de generaciones por hilo:

$$Generaciones\ por\ Hilo = \frac{Generaciones}{hilos}$$

- Por cada generación por hilo se hará de manera paralela:
  - Selección por Torneo
  - Cruce
  - Mutación
  - Medir Fitness
  - Escoger el mejor Individuo

Así, el orden final mediante esta filosofía saldrá:

$$\Theta\ (TamPoblacion * (np + ng^2 * na^2 * np)$$

$$+$$

$$Tiempo\ Generación\ Hilos\ (OMP)$$

$$+$$

$$\frac{Generaciones}{hilos} * (TamPoblacion * (ln(TamPoblacion) + ng^2 * na^2 * np))$$

$$+$$

$$Barrera\ sincronización\ final\ (OMP)$$

$$)$$

6.1.3. MPI.

Suponiendo el tiempo secuencial como $t(n)$ lo que se intenta es reducir el orden de ejecución en función del número de cores: $\frac{t(n)}{procesos}$

$$Generaciones\ por\ Proceso = \frac{Generaciones}{procesos}$$

Con la misma lógica de división entre procesos, pero, sin embargo, aquí tenemos que tener en cuenta el costo del envío de mensajes.

El proceso 0 leerá de la entrada estándar y comunicará a todos los procesos los parámetros leídos. Dados:

- $ts\ (tiempo\ inicialización\ para\ el\ envío)$
- $tw\ (tiempo\ envío\ dato)$

Tenemos al principio dos mensajes broadcast.

Cada 25% de generaciones se hace una comunicación de los mejores individuos.

Comunicación final para enviar los mejores fitness al proceso 0.

$$\Theta\ (Tiempo\ Gestión\ (MPI)$$

$$+$$

$$2 * ts * tw\ \textit{(comunicación inicial)}$$

$$+$$

$$TamPoblacion * (np + ng^2 * na^2 * np)$$

$$+$$

$$\frac{Generaciones}{procesos} * (TamPoblacion * (ln(TamPoblacion) + ng^2 * na^2 * np))$$

$$\frac{Generaciones}{procesos} * 0.25\% * \ 2 * ts * tw\ \textit{(comunicación entre generaciones)}$$

$$+$$

$$ts * tw\ \textit{(comunicación final)}$$

$$)$$

6.1.3. MPI + OpenMP.

Suponiendo el tiempo secuencial como $t(n)$ lo que se intenta es reducir el orden de ejecución en función del número de cores: $\frac{t(n)}{cores}$

Se pretende crear un número de procesos igual a la mitad de los cores y aprovechar la otra mitad para los hilos dentro de los procesos.

De este modo, la mitad de los cores serán ocupados por procesos y la otra mitad en hilos (*cuando se trabaje en una sola máquina*).

Así, el problema de subdivide dos veces y se puede trabajar de manera independiente un mayor amplio de poblaciones de manera simultánea.

$$\Theta\ (Tiempo\ Gestión\ (MPI)$$

$$+$$

$$2 * ts * tw \textit{ (comunicación inicial)}$$

$$+$$

$$TamPoblacion * (np + ng^2 * na^2 * np)$$

$$+$$

$$\frac{Generaciones}{procesos} * \frac{1}{hilos} * (TamPoblacion * (ln(TamPoblacion) + ng^2 * na^2 * np))$$

$$+$$

$$ts * tw \textit{ (comunicación final)}$$

$$)$$

6.2. Speedup.

$$S(n,p) = \frac{t(n)}{t(n,p)}$$

El speedup teórico conseguido para **OMP** sería el cociente entre el tiempo secuencial y el tiempo OMP. Vemos que difieren en que se paraleliza la parte gruesa del algoritmo en un factor de p (*nº de cores en este caso*). Sin embargo, se añade un sobrecargo por el manejo de los hilos.

El speedup teórico conseguido para **MPI** sería parecido, pero hay que tener en cuenta también el sobreesfuerzo por la comunicación que se deben hacer entre los procesos. De este modo, podemos ver que, de nuevo, se paraleliza la parte gruesa del algoritmo en un factor de p (*nº de procesos en este caso*) pero sin embargo tenemos la vejez de las comunicaciones y sus barreras implícitas.

El speedup teórico conseguido para **MPI+OMP** sería viendo la coalición de los beneficios y desventajas que nos aportan ambas tecnologías. se paraleliza la parte gruesa del algoritmo en un factor de p (*nº de cores en este caso*). Nótese que en este caso se abarca un mayor espectro, pero puede que exista una sobrecarga en un mismo equipo. Si las comunicaciones son pocas y pequeñas, sería muy rentable enviar X procesos a ciertas máquinas distintas y que utilicen todos sus cores para los hilos; tendríamos un rendimiento fantástico.

6.3. Eficiencia.

Es el **speed-up** partido por el número de elementos.

Sin embargo, como no hemos calculado el valor concreto del speed-up para cada caso no podemos ver el valor exacto de la eficiencia.

$$E(n,p) = \frac{S(n,p)}{p}$$

6.5. Escalabilidad.

La idea es, para un determinado programa, ser capaces de determinar si en el futuro, cuando aumente el número de elementos de proceso y se quiera resolver problemas mayores, se seguirán manteniendo las prestaciones. Podemos calcularlo con la función de Isoeficiencia:

$$t(n) = \frac{E(n,p)}{1-E(n,p)} t_o(n,p) = K t_o(n,p)$$

7. Estudio experimental.

Para el estudio experimental se harán pruebas de las soluciones propuestas modificando:

- Número de personas (*np*).
- Número de grupos (*ng*).
- Número de asignaturas (*na*).
- Número de cores del sistema.

Disponemos a analizar las posibles mejoras en el tiempo y se compararán las soluciones obtenidas para ver si mediante alguna de las técnicas obtenemos una mejor bondad en los resultados.

Se realizan las pruebas en:

1. Un equipo Linux con 4 cores a 4.6 GHz.
2. Jupiter con 12 cores físicos + *HyperThreading* a 1.2 GHz (según lscpu)

Por lo tanto, la tabla de pruebas con las variaciones mencionadas es:

np	**ng**	**na**	**cores**
9	3	4	4 / 24
18	6	8	4 / 24
36	12	16	4 / 24
150	20	30	4 / 24
1000	40	80	4 / 24

7.1. Versión secuencial.

Para el equipo Linux con 4 cores a 4.6 GHz tenemos:

Problema	Tiempo (segundos)	Fitness
9/3/4	2.56	1.375
18/6/8	5.57	2
36/12/16	15.12	3.21875
150/20/30	46.23	5.24
1000/40/80	351.24	10.3209

Para Júpiter con 24 cores lógicos a 1.2 GHz tenemos:

Problema	Tiempo (segundos)	Fitness
9/3/4	5.29	1.375
18/6/8	11.53	2.21875
36/12/16	31.61	3.30469
150/20/30	98.62	5.34667
1000/40/80	773.14	11.3412

7.2. Versión OpenMP.

Para el equipo Linux con 4 cores a 4.6 GHz tenemos:

Problema	Tiempo (segundos)	Fitness
9/3/4	0.82	1.375
18/6/8	1.74	2
36/12/16	4.63	3
150/20/30	13.57	5.18
1000/40/80	94.44	10.2962

Para Júpiter con 24 cores lógicos a 1.2 GHz tenemos:

Problema	Tiempo (segundos)	Fitness
9/3/4	1.42	1.375
18/6/8	2.17	2
36/12/16	3.95	3.21875
150/20/30	12.16	6.57778
1000/40/80	77.57	16.2112

7.3. Versión MPI.

Para el equipo Linux con 4 cores a 4.6 GHz tenemos:

Problema	Tiempo (segundos)	Fitness
9/3/4	0.77	1.375
18/6/8	1.5	2
36/12/16	4.00	3
150/20/30	12.00	5.12889
1000/40/80	89.82	10.2687

Para Júpiter con 24 cores lógicos a 1.2 GHz tenemos:

Problema	Tiempo (segundos)	Fitness
9/3/4	0.42	1.375
18/6/8	0.94	2
36/12/16	2.32	3.11719
150/20/30	7.34	6.52
1000/40/80	56.74	16.1222

7.5. Versión MPI + OpenMP.

Para el equipo Linux con 4 cores a 4.6 GHz tenemos:

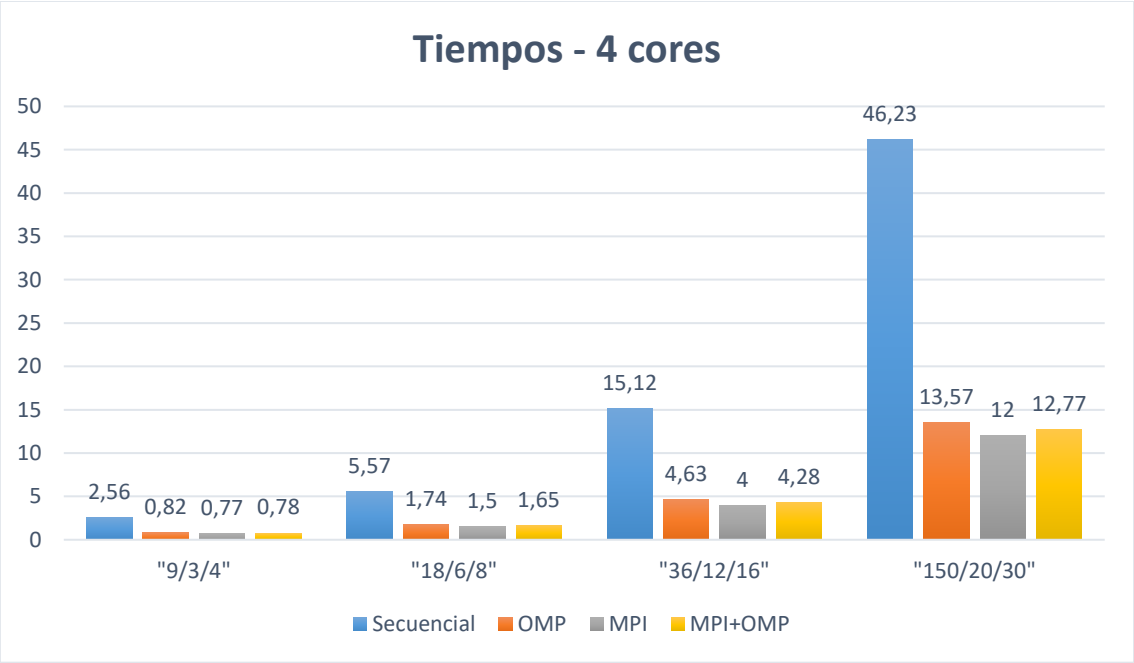
Problema	Tiempo (segundos)	Fitness
9/3/4	0.78	1.375
18/6/8	1.65	2
36/12/16	4.28	3
150/20/30	12.77	5.2
1000/40/80	91.91	10.2962

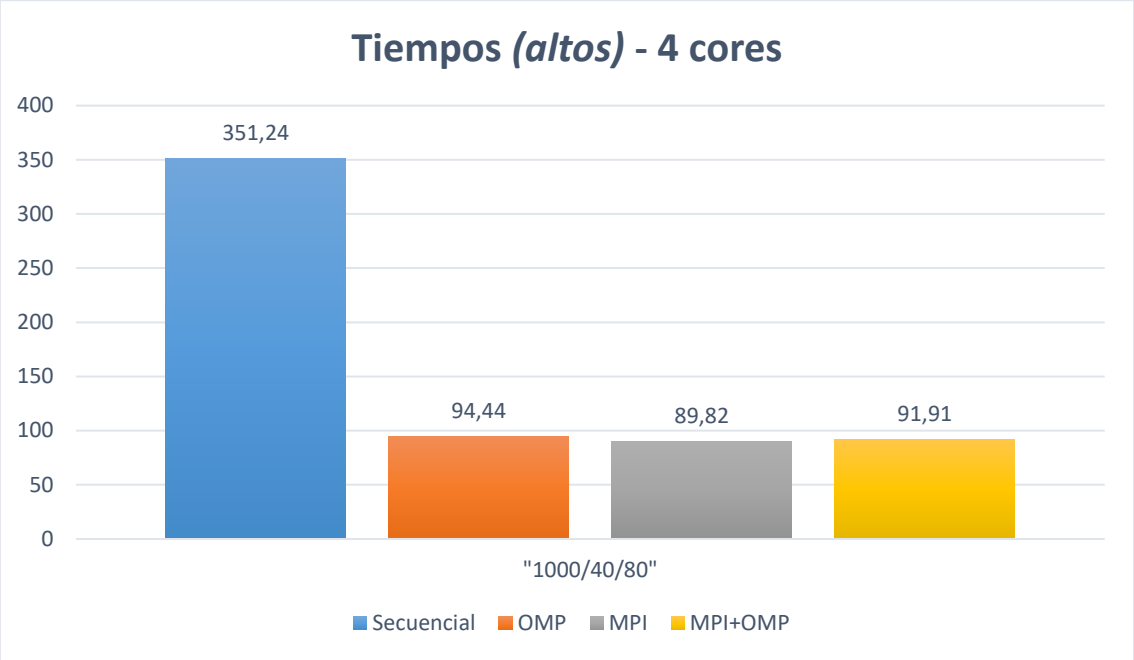
Para Júpiter con 24 cores lógicos a 1.2 GHz tenemos:

Problema	Tiempo (segundos)	Fitness
9/3/4	0.51	1.375
18/6/8	0.93	2
36/12/16	2.45	3.21875
150/20/30	7.39	6.51778
1000/40/80	56.72	16.16

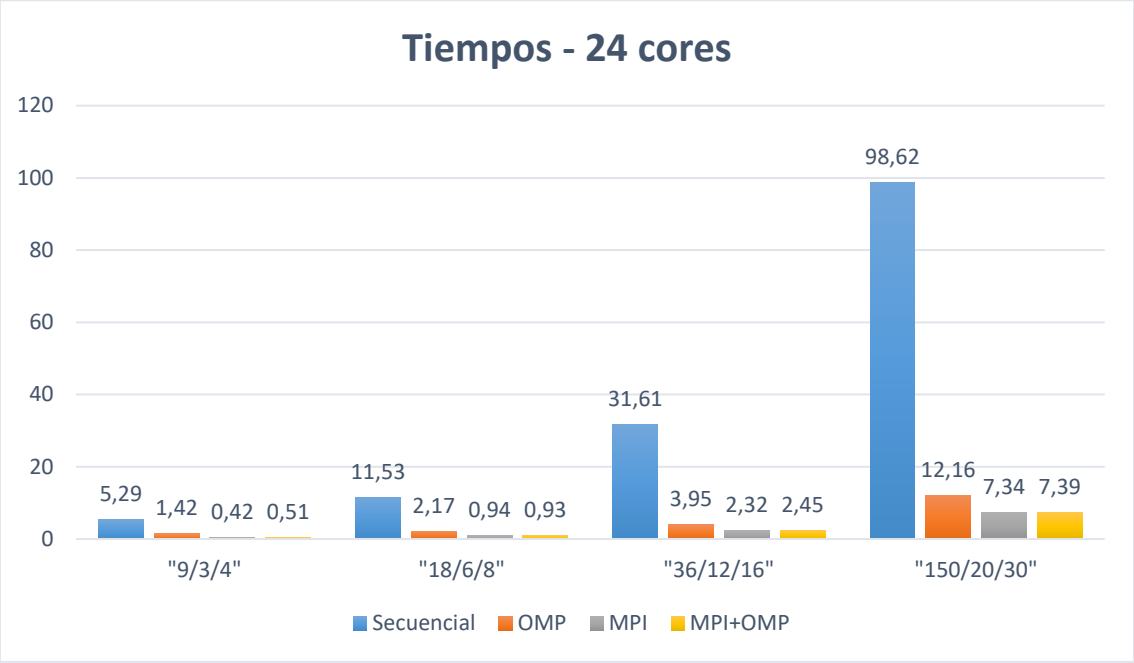
7.6. Tiempos de ejecución.

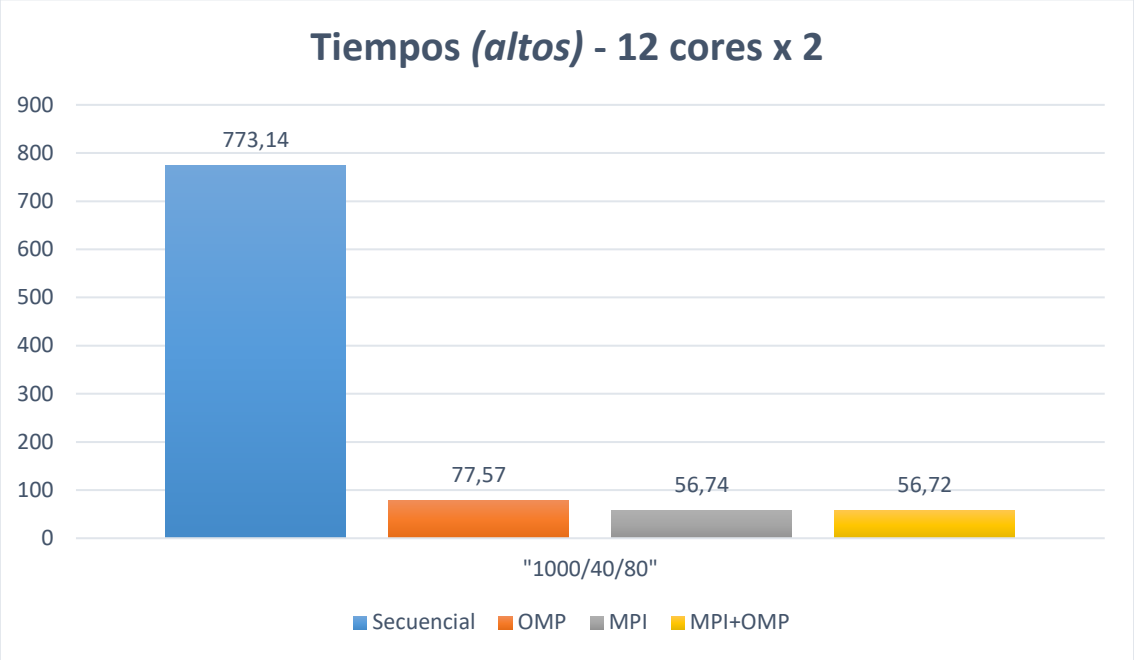
7.6.1. Equipo con 4 cores.





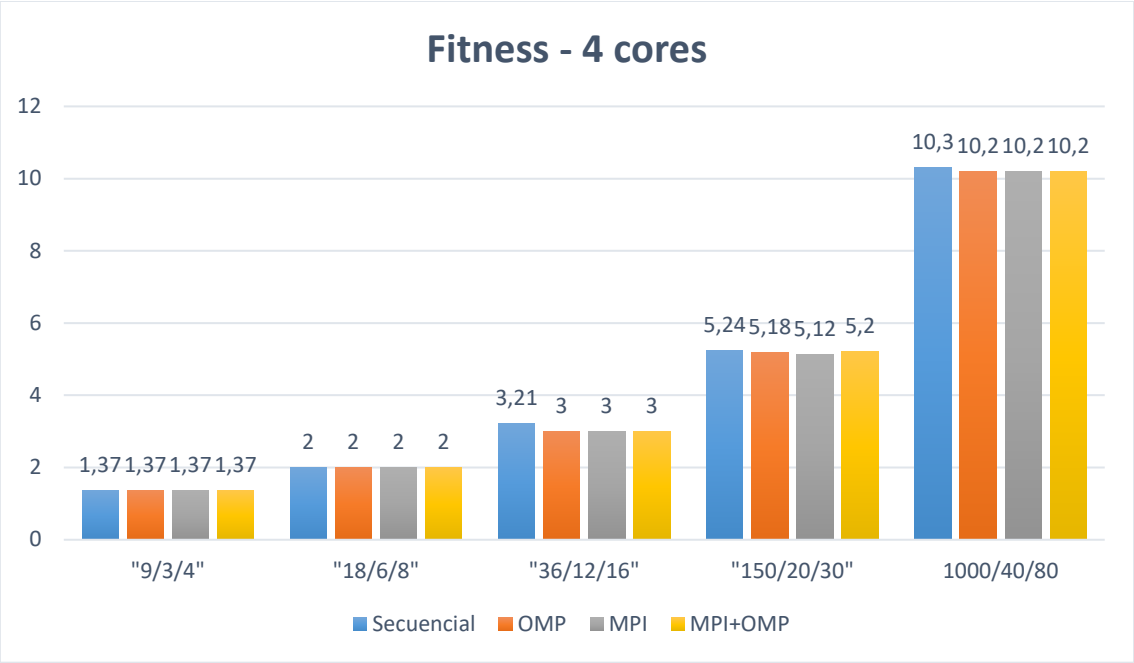
7.6.2. Equipo con 24 cores.



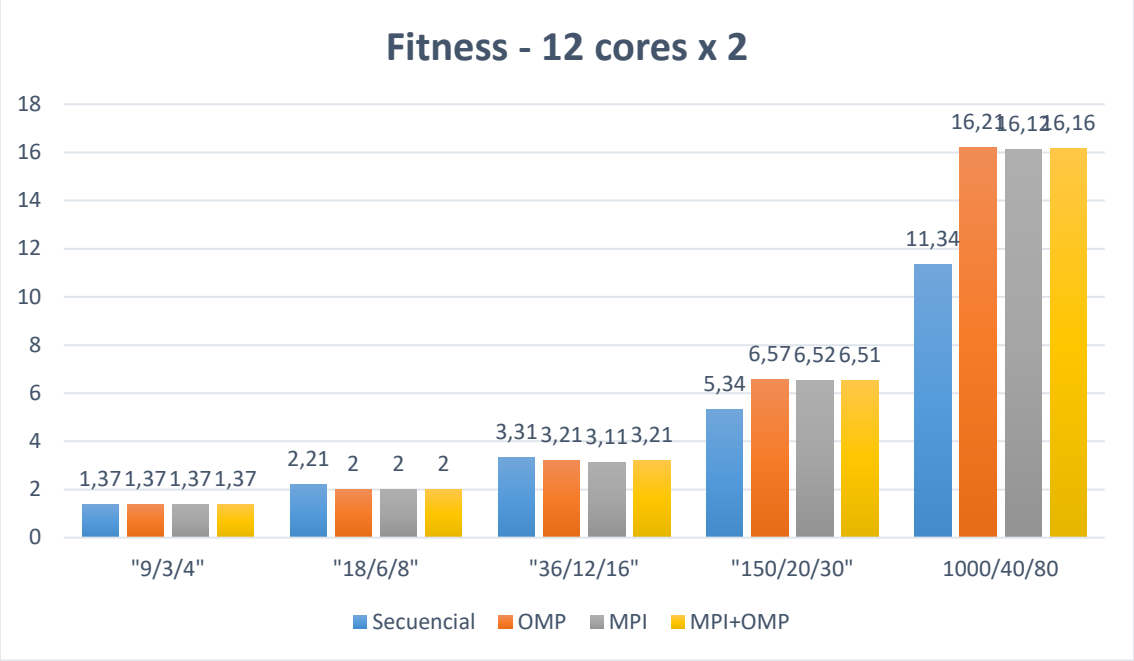


7.7. Bondad de la solución.

7.7.1. Equipo con 4 cores.



7.7.2. Equipo con 24 cores.



7.8. Análisis de los resultados obtenidos.

Cogeremos como referencia el resultado del problema con el mayor tamaño, ya que parece el más representativo y, de este modo, podemos comparar con las vejeces de cada tecnología con una amplia cantidad de datos.

7.8.1. Speedup.

$$speedup = \frac{tiempo\ secuencial}{tiempo\ paralelo}$$

Sistema Linux con 4 cores

Speedup	Representación	Valor
OMP	351.24 / 94.44	3.72
MPI	351.24 / 89.82	3.91
MPI+OMP	351.24 / 91.91	3.82

Sistema Júpiter con 12 cores x 2

Speedup	Representación	Valor
OMP	773.14 / 77.57	9.96
MPI	773.14 / 56.74	13.6
MPI+OMP	773.14 / 56.72	13.6

7.8.2. Eficiencia.

Supones que los cores por *HyperThreading* son elementos computacionales.

$$eficiencia = \frac{speedup}{elementos\ computacionales}$$

Sistema Linux con 4 cores

Speedup	Representación	Valor
OMP	3.72 / 4	0.93
MPI	3.91 / 4	0.98
MPI+OMP	3.82 / 4	0.96

Sistema Júpiter con 12 cores x 2

Speedup	Representación	Valor
OMP	9.96 / 24	0.42
MPI	13.6 / 24	0.57
MPI+OMP	13.6 / 24	0.57

7.8.3. Escalabilidad.

Como podemos observar, la escalabilidad que obtenemos para el equipo de 4 cores es directa. Sin embargo, cuando comparamos con el equipo de 12 cores por dos hilos cada core, vemos que se acerca más a una eficiencia de 12 cores. Esto puede ser debido a dos causas:

1. Que nuestra escalabilidad vaya decayendo con los números de cores
2. Que, en realidad, el equipo o el problema no funciona muy bien con procesadores con *SMT* o *HyperThreading*.

7.8.4 Conclusiones.

Como podemos observar, todas las técnicas utilizadas mejoran mucho el resultado secuencial. Cabría destacar que OMP no funciona muy bien con grandes cantidades de datos, imagino, por la compartición que tiene que tener de los datos de entrada.

Por otro lado, es cierto que para MPI hay que comunicar los parámetros los cuales se replican en todos los procesos, pero eso es beneficioso ya que acceden de manera exclusiva a ellos.

Respecto a las comparaciones de los algoritmos paralelos entre sí, se puede ver que utilizar MPI y realizar comunicaciones entre procesos requiere un coste mayor en tiempo de ejecución, sobre todo con tamaños pequeños, que podemos ver que el algoritmo con OpenMP (y más tarde el híbrido MPI+OPM) se ejecuta mucho más rápido. Sin embargo, se puede observar que el impacto del coste de las comunicaciones disminuye en gran medida, así como se va incrementando el tamaño de la entrada.

Echando un vistazo a las gráficas podemos ver que cualquiera de las implementaciones es mejor que la versión secuencial, con lo que se podría decir que son unos buenos programas. La versión MPI es la que en general consigue de las mejores soluciones, rivalizando en algunos momentos con la versión híbrida, consiguiendo soluciones similares en la mayor parte de los casos.

La bondad de las soluciones se mantiene en todas las técnicas y en todas las ejecuciones. Casi siempre obtenemos el mismo fitness: reducimos el tiempo de ejecución. Por ello no hay necesidad de dedicar más tiempo a la búsqueda ya que con técnicas más rápidas obtenemos casi el mismo resultado.

Nótese, sin embargo, que la ejecución secuencial del último problema (que tardó 13 minutos), da un resultado fitness un 42% mejor que los paralelos, pero el resultado paralelo se obtiene mucho más rápido (aproximadamente un minuto).

8. Explotación mediante *Heretosolar*

Se propone ver qué explotación podemos obtener mediante las versiones *MPI* y *MPI+OpenMP*.

Vamos a ejecutar el test más grande "*1000/40/80*" utilizando todos los equipos que podamos.

Por un lado, para *MPI* vamos a ejecutar tantos procesos como unidades de computación tengamos.

Para *MPI+OpenMP*, por otro lado,vamos a optar por utilizar la mitad para los procesos y la otra mitad para los hilos dentro de cada proceso..

Enviamos a *Júpiter, Saturno, Venus, Mercurio y Marte.*

El número total de unidades de computación es $24 + 24 + 12 + 6 + 6 = \mathbf{72}$

8.1. MPI.

```
mpirun -np 24 -host jupiter --prefix $MPI_HOME ./mpiopenmp.out < test_05.txt : \
-np 24 -host saturno --prefix $MPI_HOME ./mpiopenmp.out < test_05.txt : \
-np 12 -host venus --prefix $MPI_HOME ./mpiopenmp.out < test_05.txt : \
-np 6 -host mercurio --prefix $MPI_HOME ./mpiopenmp.out < test_05.txt : \
-np 6 -host marte --prefix $MPI_HOME ./mpiopenmp.out < test_05.txt
```

Tiempo obtenido es de: 19.88 segundos

Fitness obtenido es de: 17.15

8.2. MPI + OpenMP

```
mpirun -np 12 -host jupiter --prefix $MPI_HOME ./mpiopenmp.out < test_05.txt : \
-np 12 -host saturno --prefix $MPI_HOME ./mpiopenmp.out < test_05.txt : \
-np 6 -host venus --prefix $MPI_HOME ./mpiopenmp.out < test_05.txt : \
-np 3 -host mercurio --prefix $MPI_HOME ./mpiopenmp.out < test_05.txt : \
-np 3 -host marte --prefix $MPI_HOME ./mpiopenmp.out < test_05.txt
```

Tiempo obtenido es de: 19.02 segundos

Fitness obtenido es de: 17.27