Informe Técnico: Paralelización con OpenMP

1. Índice

---

2. Introducción

El proyecto básicamente consta en paralelizar dos aplicaciones en C. Contamos con el compilador gcc. La aplicación está enfocada a los sistemas multiprocesador (SMP) y utilizaremos la interfaz 'OpenMP' para la paralelización. Para ello crearemos una nueva versión 'paralela' donde aplicaremos la paralelización, con el objetivo de acelerar la versión en serie.

---

3. Fundamentos Teóricos

Los conocimientos esenciales que se deben considerar al iniciar este proyecto en relación con la paralelización son los siguientes:

- Al ejecutar un programa de forma paralela, existen secciones del código que deben ejecutarse de manera secuencial. Si el tamaño de la sección secuencial aumenta al mismo ritmo que el problema, el potencial de paralelización se verá limitado. En contraste, si el tamaño de la sección secuencial no se ve afectado por el aumento del tamaño del problema, el paralelismo se utilizará eficazmente para resolver problemas de gran envergadura.
- En sistemas multiprocesador, los procesos paralelos emplean variables compartidas para acceder a la memoria.

Por esta razón, se recurre a procesos para sincronizar estas variables, como las secciones críticas: fragmentos del código paralelo ejecutados por todos los procesadores de manera secuencial, y las barreras: puntos específicos del programa en los cuales los procesos esperan hasta que todos han alcanzado ese mismo punto.

- Al distribuir el trabajo en una sección paralela, el uso de un reparto estático garantiza que no se producirán sobrecargas en la ejecución de los procesos, aunque no asegura una distribución equitativa del trabajo.

Por ello, se emplea un reparto dinámico, que involucra dos variables: el chunk scheduling, donde todas las iteraciones se reparten en bloques del mismo tamaño (priorizando el tiempo de compilación), y el guided-scheduling, donde los bloques no son todos del mismo tamaño, comenzando con bloques que tienen un gran número de iteraciones y disminuyendo progresivamente (priorizando el tiempo de ejecución).

4. La versión paralela de los programas

Programa 1: integral.c

Snippet de la parte a paralelizar:

```c

```
double Integrar (double a, double b, int n, double w) {
  double  resultado, x;
  int     i;
  resultado = (f(a) + f(b)) / 2.0;
  #pragma omp parallel for private(x) reduction(+:resultado)
  for (i=1; i<n; i++) {
    x = a + i*w;
    resultado += f(x);
  }
  resultado *=  w;
  return (resultado);
}
```

Descripción del programa:

El programa Integral.c implementa el cálculo de la integral de una función mediante la regla del trapecio. La función `f(x)` a integrar es definida en el código, y se utiliza el método de la serie para mejorar la precisión del resultado. Se han incorporado clausulas de OpenMP para paralelizar el bucle principal y acelerar así el proceso de cálculo.

Elementos Relevantes:

- **Directivas OpenMP:**
    - `#pragma omp parallel for`: Se utiliza para paralelizar el bucle principal de cálculo de la integral, distribuyendo las iteraciones entre los hilos disponibles.
    - `reduction(+:resultado)`: Garantiza la operación segura de reducción para la variable `resultado`.
- **Optimización de Rendimiento:**

- La paralelización del bucle permite la ejecución simultánea de iteraciones, mejorando la eficiencia computacional.

Resultados y Tiempos:

Resultados para Integral.c:

Hilos	Tiempo (s)
1	2.17
2	0.57
4	0.28
8	0.14
16	0.07
32	0.04

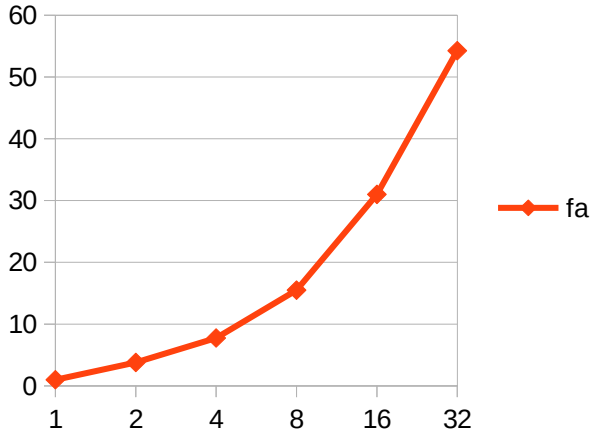


Figure 1: factor de aceleracion (fa)

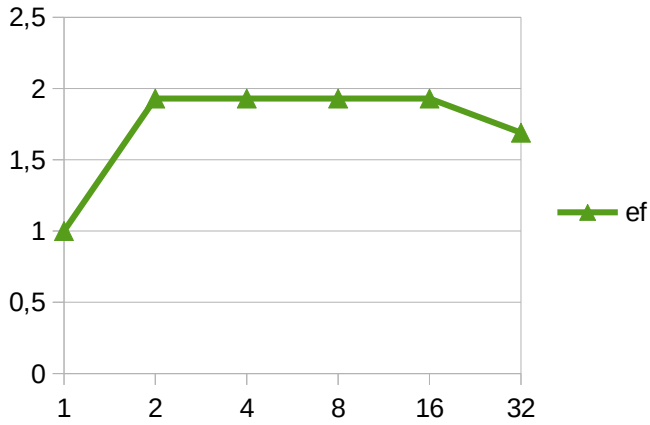


Figure 2: eficiencia

Al analizar los datos proporcionados, se evidencia un comportamiento interesante en el rendimiento en paralelo en relación con el número de hilos utilizados. A medida que se incrementa el número de hilos, se observa una disminución en el tiempo de ejecución hasta llegar a 32 hilos, es decir que si lo ejecutasemos con mas hilos hay una posibilidad de que disminuya algo más el tiempo.

La representación gráfica del factor de aceleración revela un aumento constante hasta 32 hilos. La eficiencia, reflejada en la gráfica correspondiente, muestra un rendimiento óptimo con 2 hilos, pero se mantiene a medida que se incrementa el número de hilos, hasta que con 32 hilos baja, indicando una escalabilidad débil.

En términos de los mejores tiempos de ejecución en paralelo, se lograron con 32 hilos, con un tiempo máximo de 0:00.9 segundos, mínimo de 0:00.03 segundos, media de 0:00.04 segundos y una desviación típica de 0:00.03 segundos.

En conclusión, los resultados sugieren que el sistema exhibe una escalabilidad débil, ya que el aumento de hilos sin incrementar el tamaño del problema (N) resulta en una disminución de la eficiencia. Sería recomendable considerar el tamaño del problema al incrementar el número de hilos para mantener la eficiencia del sistema. Los mejores resultados en tiempo de ejecución se obtuvieron con 32 hilos, aunque se debería tener en cuenta la escalabilidad para un rendimiento óptimo en otras configuraciones.

Programa 2: pitagoras.c

Snippet de la parte a paralelizar:

```c
double Integrar (double a, double b, int n, double w) {
  double  resultado, x;
  int     i;
  resultado = (f(a) + f(b)) / 2.0;
  #pragma omp parallel for private(x) reduction(+:resultado)
  for (i=1; i<n; i++) {
    x = a + i*w;
    resultado += f(x);
  }
  resultado *=  w;
  return (resultado);
}
```

Descripción del programa:

El programa Pitagoras.c busca y verifica pares de números cuya suma sea un cuadrado perfecto. Se han implementado clausulas de OpenMP para paralelizar la búsqueda de estos pares y mejorar la eficiencia del programa.

Elementos Relevantes:

- **Directivas OpenMP:**
  - `#pragma omp parallel for`: Paraleliza la generación de cuadrados iniciales.
  - `schedule(static)`: Distribuye las iteraciones del bucle de búsqueda de manera estática entre los hilos.
- **Reducción y Sección Crítica:**
  - `reduction(+:count)`: Asegura la operación segura de reducción para la variable `count`.
  - `#pragma omp critical`: Garantiza la exclusión mutua al imprimir los pares encontrados.

Resultados y Eficiencia:

Resultados para Pitagoras.c (N = 100):

Hilos	N	Tiempo (s)
1	100	0m0.002s
1	1000	0m0.006s
1	5000	0m0.118s
1	10000	0m0.466s
1	100000	0m13.612s
2	100	0m0.002s
2	1000	0m0.005s
2	5000	0m0.090s
2	10000	0m0.349s
2	100000	0m9.472s
4	100	0m0.002s
4	1000	0m0.005s
4	5000	0m0.053s
4	10000	0m0.207s
4	100000	0m5.336s
8	100	0m0.002s
8	1000	0m0.003s
8	5000	0m0.029s
8	10000	0m0.112s
8	100000	0m2.830s
16	100	0m0.002s
16	1000	0m0.004s
16	5000	0m0.017s
16	10000	0m0.059s
16	100000	0m1.465s
32	100	0m0.003s
32	1000	0m0.004s
32	5000	0m0.013s
32	10000	0m0.032s
32	100000	0m0.748s

Como este programa lo ejecutamos con distintos valores de N mostraré el factor de aceleración mediante una tabla y así no tener 5 graficas diferentes.

Hilos	N=100	N=1000	N=5000	N=10000	N=100000
1	1	1	1	1	1
2	1.583	1.2	1.311	1.336	1.435
4	2.609	1.371	2.226	2.254	2.544

8	5.92	2.227	4.069	4.161	4.677
16	8.706	2.431	6.941	7.898	9.291
32	10.571	2.626	9.077	14.188	18.192

Al analizar los datos teniendo en cuenta la variación en el tamaño del problema (N) para cada configuración de hilos en el programa "pitagoras.c", se observa que para algunos valores específicos de N, ciertos números de hilos ofrecen un mejor rendimiento. En particular:

- Para N=100, el mejor rendimiento se logra con 8 hilos.
- Para N=1000, 16 hilos muestran un mejor rendimiento.
- Para N=5000, 32 hilos proporcionan un rendimiento superior.
- Para N=10000, nuevamente 32 hilos son óptimos.
- Para N=100000, se obtienen los mejores resultados con 32 hilos.

Estos hallazgos indican que la elección del número de hilos óptimo depende del tamaño específico del problema, y se destaca la importancia de ajustar la configuración de hilos de manera adecuada para cada situación particular. Aunque se observa una variabilidad en el rendimiento, la adaptación del número de hilos según el tamaño del problema puede mejorar la eficiencia global del programa "pitagoras.c".

El factor de aceleración revela patrones fluctuantes, destacando la importancia de considerar cuidadosamente la escalabilidad del programa en diferentes configuraciones.

Programa 3: reparto_it.c

Snippet de la parte a paralelizar:

```c

```
 // Bucle 1 para paralelizar: reparto estatico consecutivo
#pragma omp parallel for private(i, tid)
 for (i=0; i<N; i++) {
   tid = omp_get_thread_num();
   fun (i);
   A[i] = tid;
 }
 // Bucle 2 para paralelizar: reparto dinamico, chunk scheduling
 // lote (chuck) de 6 iteraciones (LOTE)
#pragma omp parallel for private(i, tid) schedule(dynamic, LOTE)
```

```
  for (i=0; i<N; i++) {
    tid = omp_get_thread_num();
    fun (i);
    B[i] = tid;
  }
```

Descripción:

El programa Reparto_it.c realiza dos bucles, cada uno con un método de reparto diferente. El primero utiliza un reparto estático consecutivo, y el segundo utiliza un reparto dinámico con un tamaño de lote específico (6 en este caso). Ambos bucles se paralelizan para mejorar la eficiencia del programa.

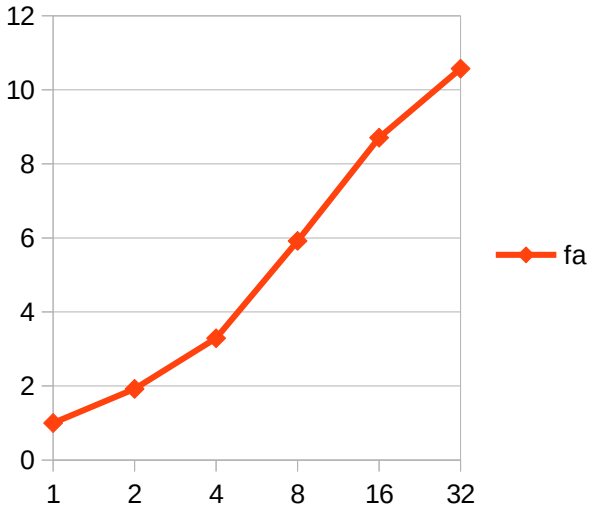
Elementos Relevantes:

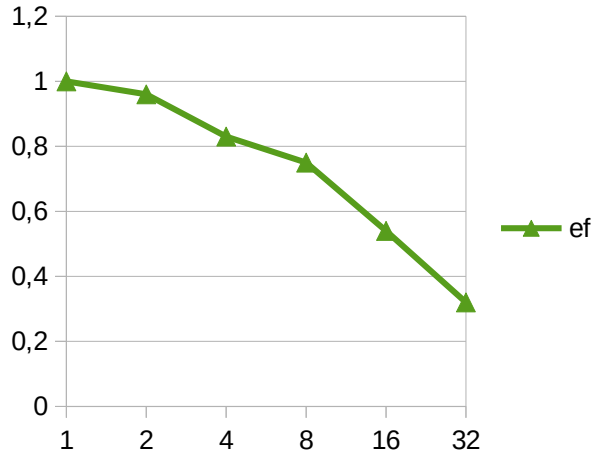
- **Directivas OpenMP:**
  - `#pragma omp parallel for`: Paraleliza ambos bucles para distribuir las iteraciones entre los hilos.
  - `private(i, tid)`: Declara variables privadas para evitar condiciones de carrera.
  - `schedule(dynamic, LOTE)`: Aplica un reparto dinámico con un tamaño de lote definido.
- **Función de Espera:**
  - `usleep(100 * (x % 15))`: Introduce una espera artificial para simular carga y destacar el reparto dinámico.

Visualización de Resultados:

Resultados para Reparto_it.c:

Hilos	Tiempo (s)
1	0.148
2	0.077
4	0.045
8	0.025
16	0.017
32	0.014





Al analizar los datos proporcionados de la ejecución del programa "repartos_it.c", se observa un patrón interesante en el rendimiento en paralelo en función del número de hilos utilizados. A medida que se incrementa el número de hilos, se evidencia una disminución en el tiempo de ejecución, alcanzando su punto más bajo con 32 hilos.
La representación gráfica del factor de aceleración revela un aumento constante hasta 32 hilos, indicando un mejor rendimiento a medida que se emplean más recursos paralelos. Sin embargo, es crucial considerar la eficiencia del sistema al evaluar estos resultados.

La eficiencia, reflejada en la gráfica correspondiente, muestra un rendimiento óptimo con 1 hilo, pero disminuye a medida que se incrementa el número de hilos, sugiriendo una escalabilidad que puede no ser completamente eficiente.

En términos de los mejores tiempos de ejecución en paralelo, se lograron con 32 hilos, con un tiempo máximo de 0m0.217s, mínimo de 0m0.014s, media de 0m0.046s y una desviación típica de 0m0.058s.

En conclusión, los resultados sugieren que el sistema exhibe una mejora en el rendimiento al aumentar el número de hilos, aunque la escalabilidad puede no ser totalmente eficiente.

---

5. Conclusiones

En términos de rendimiento, la paralelización con OpenMP proporciona mejoras significativas en los tres programas. Sin embargo, es fundamental realizar un análisis detallado para optimizar el rendimiento según las características específicas de cada problema.