Cálculo aproximado del número Pi

Se partió de la base de calcular el pi bajo el concepto del área bajo la curva de una función que aproxima el valor real de esta constante, a medida de que se toman mayores rectángulos (*num_steps*) . Este problema es digno de paralelizar porque se puede resolver el área de estos rectángulos de forma paralela. En teoría, uno podría suponer que si se cuentan con tantos procesadores como rectángulos, el tiempo se reduciría al costo de calcular el área de un solo rectángulo.

La realidad es que esto no es así, ya que a nivel práctico no se cuenta con un número alto de procesadores . Aún así, si se cuenta con tantos procesadores como número de paso, el agregar más paralelismo produce mayor overhead, haciendo que en un determinado número de procesadores la performance empiece a empeorar.

Antes de explicar cada una de las etapas, es necesario aclarar que la medición de tiempos se realizó de la siguiente manera :

```
double time_start = omp_get_wtime();
//Inicializaciones y Cuerpo del Algoritmo
(...)
double time_end = omp_get_wtime();
procesar(time_end - time_start);
```

Es decir, que tanto las inicializaciones de variables como el propio algoritmo están delimitados por dos mediciones de tiempo, para ver qué tan rápido funciona.

Ahora, se procede a explicar cada una de las técnicas :

- **Secuencial** : se tomó el código propuesto en la teoría por la cátedra , y es la referencia para saber qué tan mejor o peor funcionan las diferentes técnicas de paralelismo.
- **No reduction** : detectando que el paralelismo se puede realizar a nivel de iteración, de la siguiente manera :

```
#pragma omp parallel for
for(int i = 0 ; i < n_steps ; i++){
    x = (i+ 0.5)*step;
    sum += 4.0/(1.0+x*x);
}
```

El problema acá es que *sum* es una variable compartida, y cada thread la irá modificando, y es necesario que se sincronice dicha variable. Esto se controló encerrando a dicha operación en un semáforo a través del atomic o critical : **`#pragma omp atomic | critical`**

A fines de encontrar el resultado correcto ambos funcionan. A niveles de tiempo se hizo una separación de código entre el atomic y el critical.

Se retocó el código declarando la variable x dentro del for (originariamente no era así, sino que era local al ámbito padre del for). Esto evita tener que informar en la declaración pragma que *x* es de tipo private (ya que ahora la variable es local al for). Entonces no es necesario la sentencia de más que el pragma generaría por cada instancia del for ganando un tiempo considerable que se pudo comprobar en la práctica.

- **Reduction** : para esta técnica, ya no es necesario contar con un pragma por separado que controle el sincronismo de la actualización de la variable *sum* . Lo que se hace es agregar al pragma que toma la iteración, concatenado con la operación de reduction, y a qué variable se le asocia :
  ```
  #pragma omp parallel for reduction(+:sum)
  ```
  Teóricamente, dicha técnica mejora respecto de no aplicar reducción.

  Aquí se fue probando el uso de variantes como compartir el número de pasos y el paso propiamente dicho, pero se decidió agregar a x como variable privada, concatenando esta información dentro del pragma:
  ```
  #pragma omp parallel for reduction(+:sum) private(x)
  ```

- **Scheduling** : la técnica de planificación sirve para administrar la forma de ir asignando a cada thread las iteraciones a procesar. Si bien teóricamente cada thread debería trabajar con (n/cant_hilos)*100 % , la realidad es que hay hilos que procesan mucho más rápido que otros, y puede que una tarea sea más larga que otra.

  La planificación se agregó a la técnica de reduction, para que mejore aún más el tiempo :
  ```
  #pragma omp parallel for schedule(static|dynamic|guided, chunk_size) reduction(+:sum)
  ```

  Se hicieron pruebas para la planificación de tipo *static* , *dynamic* y *guided* para ver si existe una diferencia notable entre estas, y respecto a la técnica del reduction sin planificación.

  Para poder hacer un análisis más exacto de cómo se comporta el algoritmo, variando cantidad de procesadores y número de *steps*, con distintas tipos de técnicas de planificación (*static, dynamic, guided*), se optó por utilizar una función de porcentaje para calcular el número de iteraciones que el *scheduling* le asigna a cada procesador, quedando así sujeto al *steps* que se elija.
  Bajo este análisis, se pudo ver que los mejores tiempos eran alcanzados cuando el porcentaje que se le ajustaba al planificador eran bien bajos (del orden del 3% al 10%). También, se observó que la diferencia entre los distintos tipos de

técnicas de planificación no eran significativas, no pudiendo encontrar una planificación que se destacara por arrojar mejores tiempos.

Multiplicación de Matrices

Dadas dos matrices A y B de NxN , el objetivo es generar una nueva matriz C que salga del producto de A y B. El algoritmo usado es el que para obtener un valor de la posición (**i** , **j**) de la matriz de salida , se hace el producto escalar entre la fila **i** de A y la columna **j** de B. El algoritmo se describe a través del siguiente código C :

```
for (int a = 0; a < size_matrix; a++) {
    for (int i = 0; i < size_matrix; i++) {
          int adder = 0;
          for (int j = 0; j < size_matrix; j++)
              adder += matrix_A[i][j] * matrix_B[j][a];
          matrix_Out[i][a] = adder;
    }
}
```

Entonces el objetivo para el contexto del trabajo, es realizar diferentes productos escalares de forma paralela. Pero como un producto escalar es una sumatoria de las diferentes posiciones de fila y columna de los vectores seleccionados, también se busca que dichas sumas se realicen en paralelo.

Al igual que en el problema del cálculo del Pi, el tiempo se mide entre las inicializaciones y el cuerpo del algoritmo, para comparar cada una de las técnicas de paralelismo, que se enuncian a continuación :

- **No reduction :** el hecho de contar con tres sentencias de iteración como el for, tienta a poner a priori tres instrucciones de `#pragma omp parallel for`.
  La realidad es que se hicieron diferentes pruebas para ver si al aplicar dicha sentencia en uno o dos *for* genera los mismos resultados, o incluso si la multiplicación da correcta. Efectivamente si solo se aplica en la primera sentencia iterativa va a funcionar igual, e incluso con mejor performance que al usar 3 pragmas. Aún así, la elegida fue la de usar dos pragmas en los primeros *for*, ya que dio un poco mejor en tiempos que al usar el global.

  La zona de exclusión mutua para este problema es la actualización del *adder*, que es la encargada de ir sumando los resultados parciales del producto escalar. Entonces, como el problema del cálculo de Pi, se usaron tanto el atomic y el critical, para comparar cuál funciona mejor de los dos.

- **Reduction:** aquí el objetivo principal es disminuir el overhead que produce la zona de exclusión mutua, haciendo que solo un thread pueda acceder a la vez

a la operación del *adder* . Entonces a la versión no reduction se elimina el pragma que hacía las veces de semáforo, y se agrega :

```
#pragma omp parallel for reduction(+:adder)
```

- **Scheduling** : la idea es similar a lo que se planteó en el problema anterior, de probar si al administrar la forma en que los threads reciben las iteraciones, mejora o no a las técnicas anteriores, además de las distintas variantes de administración que existen.

Testeo de los problemas

- Cada uno de los problemas tiene su versión sin reduction que incluye las técnicas de *critical* y *reduction* , y la versión con reduction, que cuenta con las técnicas del *reduction normal , schedule static, schedule dynamic* y *schedule guided* . Además para cada versión se ejecuta 8 veces , aumentando el número de procesadores hasta llegar al límite de la computadora en la que se esté probando, mostrando la media y el desvío estándar de cada prueba realizada con el número de procesadores de esa iteración.

- Lo que refiere al **cálculo de PI**, se fue probando con distintas entradas, tomando como umbral aquella que diera un resultado lo suficientemente exacto de la constante para tenerla en cuenta a la hora de realizar las pruebas de tiempo. El *num_steps* elegido para mostrar los resultados finales es de **100.000.000** , que a pesar de ser un número relativamente alto, las brechas temporales dan muy bajas.

  Las pruebas de este algoritmo se realizaron en un principio en Linux con un Intel Celeron con dos procesadores. Siendo que la CPU como la cantidad de hilos no mostraba un gran tiempo y escala a más paralelismo, se probó en una computadora con SO Linux, que contaba con un Intel Core i7 de 6 procesadores.

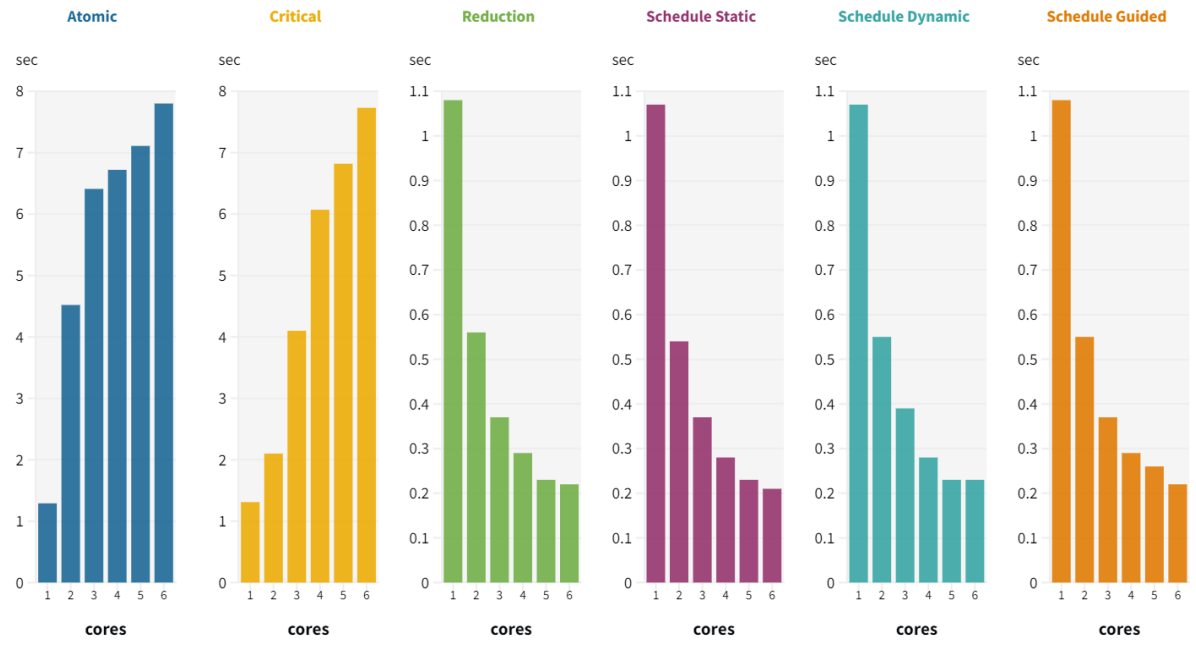
- Respecto de la **Multiplicación de Matrices**, el *size_matriz* fue tomando entradas de 500, 750 y 1000. Si bien pareciera que fueran chicas en comparación a *num_steps* , hay que considerar que la complejidad de este algoritmo es de O($size\_matriz^3$), por lo que el número de operaciones es del orden de **1.000.000.000** cuando la entrada es **1000** .

  Los testeos se realizaron en una computadora con un SO Linux, con un Intel Core i3 con 4 procesadores, y si bien se pudo observar una notable evolución entre 2 y 4 procesadores, el algoritmo tardaba en el orden de minutos con una de las técnicas que hacía lento el análisis.

Luego se probó con la misma computadora i7 del cálculo de PI, y no solo se vio la baja notable de tiempo, sino la preponderancia del overhead respecto del paralelismo a partir de 5 procesadores.

Análisis de los Resultados

Cálculo de PI

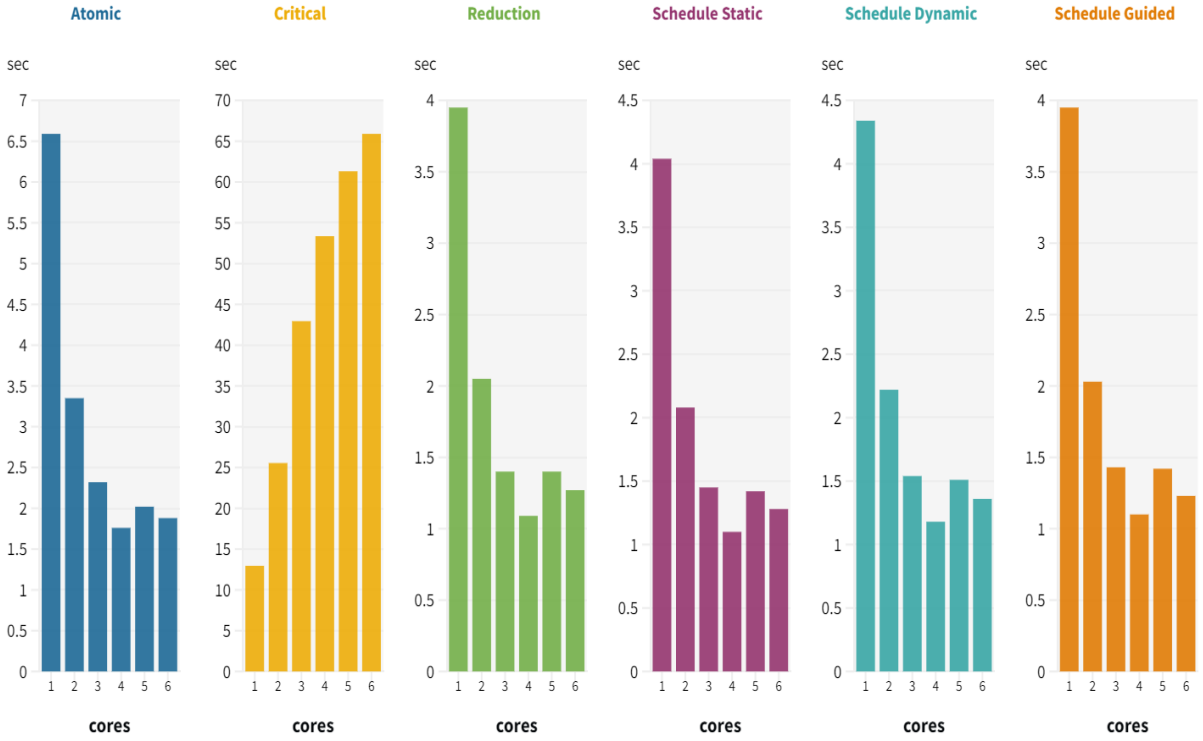


Los resultados muestran que el atomic y el critical van empeorando a medida que aumenta el número de procesadores. El motivo es que al contar con la zona de exclusión mútua al tener más hilos existe un cuello de botella por quien accede a la variable compartida *x* , y para colmo cada thread no simplifica la operación (como si haría un problema del estilo *Travelling Salesman Problem*), por lo que hace que tenga que acceder siempre al critical o atomic. Entre ellos hay una pequeña tendencia de que el atomic funciona un poco peor que el critical, pero la diferencia se achica al contar con 6 procesadores.

Respecto del reduction, se observa que los tiempos son muchos más bajos que no usarlo, esto se logra gracias a que el reduction está implementado de forma eficiente para el *for*. Está tan bien implementado que incluso al aumentar el número de procesadores el programa corre mucho más rápido. Según los resultados, para este procesador y SO, a partir de 6 procesadores, los resultados empiezan a estabilizarse.

Por el lado de la planificación, no se ven casi cambios respecto de aplicar solamente reduction, y tampoco entre las distintas técnicas de planificación. Esto sucede porque en cada iteración, las subtareas (calcular el área del rectángulo) es la misma, solo que cada paso agarra un rectángulo distinto. La planificación dinámica y guided funcionan mejor cuando existe un desbalance de carga que ellas pueden mejorar respecto del reduction normal o scheduling static.

Multiplicación de Matrices



Aquí sorprende la diferencia abismal que existe entre el critical respecto del resto, dando tiempos del orden del minuto a partir de 5 procesadores. A diferencia del problema del cálculo de Pi, acá el atomic se comporta muy bien, disminuyendo su tiempo conforme más procesadores se tenga.

El reduction posee una mejora respecto del atomic, pero no mucho. Esto demuestra que la versión sin reduction se adapta muy bien para este problema.

Con la planificación ocurre lo mismo que el problema anterior, que mejora muy poco a la técnica sin planificar, porque las tareas a asignar son del mismo tamaño.

Lo interesante que tienen en común todas las técnicas (salvo el critical) es que el mejor tiempo da con 4 procesadores, y luego tiende a subir de nuevo.