Pautas generales

Obtener el tiempo de ejecución

Para obtener el tiempo de ejecución de todos los algoritmos se debe utilizar la función provista por la cátedra (dwalltime).

Por convención sólo deberá tomarse el tiempo de ejecución del procesamiento de datos.

El tiempo de ejecución NO debe incluir:

- *Alocación y liberación de memoria.*
- *Impresión en pantalla (printf)*
- *Inicialización de estructuras de datos.*
- *Impresión y verificación de resultados.*

En el caso especial de MPI se debe tomar el tiempo de ejecución incluyendo la comunicación entre procesos.

En el caso especial de GPU se debe tomar el tiempo de ejecución incluyendo las transferencias de memoria.

Métricas de rendimiento básicas

Para comparar el rendimiento de un algoritmo paralelo respecto a su versión secuencial usaremos como métrica el Speedup. El Speedup se calcula como:

$$Speedup = \frac{Tiempo\ de\ ejecución\ Secuencial}{Tiempo\ de\ ejecución\ Paralelo}$$

Teóricamente, el Speedup máximo alcanzable es P. Donde P es el número de procesadores utilizados.

Otra métrica de rendimiento es la Eficiencia. Esta se calcula como:

$$Eficiencia = \frac{Speedup}{P}$$

Teóricamente, la Eficiencia máxima alcanzable es 1.

Práctica 1

Optimización de algoritmos secuenciales

Pautas:

En todos los ejercicios de matrices probar con tamaños de matriz potencias de 2 (32, 64, 128, 256, 512, 1024, 2048 etc.).

Compilar en Linux gcc:

gcc –o salidaEjecutable archivoFuente.c

1. *Álgebra de Matrices*

a. Analizar el algoritmo matrices.c que resuelve la multiplicación de matrices C=AB donde A, B y C son matrices cuadradas de N*N:
   i. Comparar el tiempo de ejecución original con el tiempo de eliminar las funciones getValor y setValor. ¿Son necesarias estas funciones?¿Qué puede decirse del overhead introducido por los llamados a estas funciones?
   ii. Partiendo del código sin las funciones getValor y setValor, comparar la solución que almacena A, B y C por filas con las solución que almacena B por columnas. ¿Qué conclusión puede obtenerse de la diferencia en los tiempos de ejecución? ¿Cómo se relaciona con el principio de localidad?

Ejecución: *./matrices N*

b. Analizar los algoritmos que resuelven distintas operaciones sobre matrices de N*N:

expMatrices1.c: dos versiones que resuelven la operación AB + AC + AD
expMatrices2.c: dos versiones que resuelven la operación AB + CD
expMatrices3.c: dos versiones que resuelven la operación BA + CAD

Ejecutar con distintos valores de N. Comparar los tiempos de ejecución de las dos versiones en cada caso. ¿Qué versión es más rápida? ¿Por qué?

Ejecución: *./expMatrices1 N*
./expMatrices2 N
./expMatrices3 N

c. Implementar una solución a la multiplicación de matrices AA donde la matriz A es de N*N. Plantear dos estrategias:

   1. Ambas matrices A están ordenadas en memoria por el mismo criterio (filas o columnas) .

2. La matriz A de la izquierda sigue un criterio de ordenación en memoria (por ej: filas). La matriz A de la derecha se construye (por ej: ordenada por columnas) a partir de modificar el criterio de ordenación en memoria de la matriz A izquierda. Se debe tener en cuenta el tiempo de construcción de la nueva matriz.

¿Cuál de las dos estrategias alcanza mejor rendimiento?

d. Describir brevemente cómo funciona el algoritmo multBloques.c que resuelve la multiplicación de matrices cuadradas de N*N utilizando la técnica por bloques de tamaño BS*BS. Ejecutar el algoritmo utilizando distintos tamaños de matrices y distintos tamaño de bloques, comparar los tiempos con respecto a la multiplicación de matrices optimizada del ejercicio 1. Según el tamaño de las matrices y de bloque elegido ¿Cuál es más rápido? ¿Por qué? ¿Cuál sería el tamaño de bloque óptimo para un determinado tamaño de matriz?

Ejecución: *./multBloques N BS*
BS: tamaño de bloque

e. Implementar las soluciones para la multiplicación de matrices MU, ML, UM y LM. Donde M es una matriz de N*N. L y U son matrices triangulares inferior y superior, respectivamente. Analizar los tiempos de ejecución para la solución que almacena los ceros de las triangulares respecto a la que no los almacena.

2. El algoritmo fib.c resuelve la serie de Fibonacci para un numero N dado utilizando dos métodos, el método recursivo y el método iterativo. Analizar los tiempos de ambos métodos ¿Cuál es más rápido? ¿Por qué?

Ejecución: *./fib N*
Probar con N entre 0 y 50.

3. Analizar los algoritmos productoVectorialRegistro.c y productoVectorialSinRegistro.c. Ambos programas parten de dos conjuntos de N vectores matemáticos y realizan el producto vectorial uno a uno de acuerdo al orden de cada vector en el conjunto. ¿Cuál de las dos soluciones es más rápida? ¿Por qué?:

Ejecución: ./ productoVectorialRegistro N
./ productoVectorialSinRegistro N
N tamaño de los conjuntos de vectores

4. Optimización de instrucciones

a. El algoritmo instrucciones1.c compara el tiempo de ejecución de las operaciones básicas suma (+), resta (-), multiplicación (*) y división (/), aplicadas sobre los elementos que se encuentran en la misma posición de dos vectores x e y. ¿Qué

análisis se puede hacer de cada operación? ¿Qué ocurre si los valores de los vectores x e y son potencias de 2?

Ejecución: *./instrucciones1 N r*
N: tamaño de los vectores
r: número de repeticiones

b. En función del ejercicio anterior analizar el algoritmo instrucciones2.c que aplica dos operaciones distintas a cada elemento de un vector x.

Ejecución: *./instrucciones2 N r*
N: tamaño de los vectores
r: número de repeticiones

c. El algoritmo modulo.c compara el tiempo de ejecución de dos versiones para obtener el resto de un cociente m (m potencia de 2) de los elementos enteros de un vector de tamaño N. ¿Qué análisis se puede hacer de las dos versiones?

Ejecución: *./modulo N m*
N: tamaño del vector
m: potencia de 2

5. Iteraciones

a. Analizar el algoritmo iterstruc.c que resuelve una multiplicación de matrices utilizando dos estructuras de control distintas. ¿Cuál de las dos estructuras de control tiende a acelerar el cómputo?

Compilar con la opción –O3

Ejecución: *./iterstruct N R*
N: tamaño de la matriz
R: cantidad de repeticiones

b. Analizar el algoritmo optForArray.c que inicializa un vector con sus valores en 1 de dos formas. ¿Cuál es más rápida?

Ejecución: *./optForArray N R*
N: tamaño de la matriz
R: cantidad de repeticiones

c. Analizar el algoritmo GaussFor.c que calcula la suma de N números naturales consecutivos y lo compara con la suma de Gauss.
¿Por qué la suma para N=2147483647 da resultado correcto y para N=2147483648 el resultado es erróneo? ¿Cómo lo solucionaría?

Ejecución: *./GaussFor N*
N: número de elementos a sumar

6. El algoritmo overheadIF.c da tres soluciones al siguiente problema: dado un vector V y una posición P, el algoritmo cuenta la cantidad de números del vector V que son menores al elemento en la posición P.
   Analizar los tiempos obtenidos de las tres soluciones y evaluar las fuentes de overhead en cada caso.

   Ejecución: *./overheadIF N*
   N: tamaño del vector

7. Compilar y ejecutar el archivo precision.c que calcula el número de fibonacci para los elementos de un vector de tamaño N. El algoritmo compara el resultado de aplicarlo a elementos de tipo de datos entero respecto a aplicarlo a elementos de coma flotante en simple y doble precisión.
   Analizar los tiempos obtenidos para cada tipo de datos. ¿Qué conclusiones se pueden obtener del uso de uno u otro tipo de dato?

   Compilar en simple precisión: gcc -O2 -lm -o singlep precision.c
   Compilar en doble precision: gcc -O2 -DDOUBLE -lm -o doblep precision.c

   Ejecución simple precisión: *./singlep N*
   Ejecución doble precisión: *./doblep N*
   N: tamaño del vector

8. El algoritmo porcentaje.c calcula el porcentaje de un número dado. Ejecutar el programa y verificar que los resultados son correctos:
   El 50% de 90 debería dar 45.
   El 10% de 2530 debería dar 253.
   El 90% de 20000000 debería dar 18000000.
   El 10% de 50000000 debería dar *5000000.*
   El 40% de 50000000 debería dar *20000000.*
   Analizar ¿Por qué ocurre un error al ejecutar el 50% de *50000000? ¿Cómo lo solucionaría?*

   Ejecución: *./porcentaje P N*
   P es el tanto por ciento de N

   (calcular el 50% de 90: ./porcentaje 50 90)

Práctica 2

Programación con Pthreads

Estructura general de programa para Pthreads:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* funcion(void *arg){
  int tid=*(int*)arg;
          printf("Hilo id:%d\n",tid);
         //Código que ejecutará cada hilo
          pthread_exit(NULL);
}

int main(int argc, char* argv[]){
 int T = atoi(argv[1]);
 pthread_t misThreads[T];
 int threads_ids[T];

          for(int id=0;id<T;id++){
                  threads_ids[id]=id;
                  pthread_create(&misThreads[id],NULL,&funcion,(void*)&threads_ids[id]);
          }

          for(int id=0;id<T;id++){
                  pthread_join(misThreads[id],NULL);
          }

          return 0;
}
```

NOTA: La variable T recibe el número de hilos como primer parámetro de programa. Por lo tanto, si queremos ejecutar el programa para que cree 8 hilos la línea de comando será:

```
./programa 8
```

Comentarios sobre programas que ejecutan varias etapas

Es común que los algoritmos paralelos tengan varias etapas de ejecución. NO deberían crearse Hilos una y otra vez por cada una de las etapas. Los Hilos deberán crearse una única vez y cada etapa debe estar separada por una barrera de sincronización.

En este caso, la función que implementa el comportamiento de los Hilos tendrá la siguiente forma:

```
void* funcion(void *arg){
  int tid=*(int*)arg;

        Etapa1
        Barrera de Sincronización
        Etapa2
        Barrera de Sincronización
        ...
        EtapaN
        Barrera de Sincronización

         pthread_exit(NULL);
}
```

Pautas:

Compilar en Linux gcc:

gcc -pthread –o salidaEjecutable archivoFuente

Calcular el speedup y la eficiencia del algoritmo paralelo respecto del algoritmo secuencial.

1. Paralelizar la multiplicación de matrices cuadradas de N*xN*. Obtener el tiempo de ejecución para N=512, 1024 y 2048. Ejecutar con 2 y 4 threads.

2. Paralelizar un algoritmo que cuente la cantidad de veces que un elemento X aparece dentro de un vector de N elementos enteros. Al finalizar, la cantidad de ocurrencias del elemento X debe quedar en una variable llamada *ocurrencias*. Ejecutar con 2 y 4 threads.

3. Paralelizar la búsqueda del mínimo y el máximo valor en un vector de N elementos. Ejecutar con 2 y 4 Threads.

4. Paralelizar la ordenación por mezcla de un vector de N elementos. Ejecutar con 2 y 4 Threads.

5. Paralelizar un algoritmo que calcule el valor promedio de N elementos almacenados en un vector de tamaño N. Ejecutar con 2 y 4 Threads.

6. Paralelizar un algoritmo que determine si un vector de N elementos es monotónico. Un vector es monotónico si todos sus elementos están ordenados en forma creciente o decreciente.

7. Dado un vector de N elementos enteros creciente, paralelizar un algoritmo que determine cada cuanto se duplica cada número de la serie. (-1 si no se duplica)

Práctica 3

Programación con OpenMP

Pautas:

Compilar en Linux gcc:

gcc -fopenmp –o salidaEjecutable archivoFuente

1. El programa ejercicio1.c inicializa una matriz de NxN de la siguiente manera: A[i,j]=i*j, para todo i,j=0..N-1. Compilar y ejecutar. ¿Qué problemas tiene el programa? Corregirlo.

   Ejecución: *./ejercicio1 N cantidadDeThreads*

2. Analizar y compilar el programa ejercicio2.c. Ejecutar varias veces y comparar los resultados de salida para diferente número de threads ¿Cuál es el problema? ¿Es posible corregirlo?.

   Compilar utilizando la opción –lm:

   gcc -lm -fopenmp –o salidaEjecutable archivoFuente

   Ejecución: *./ejercicio2 N cantidadDeThreads*

3. El programa matrices.c realiza la multiplicación de 2 matrices cuadradas de N*N (C=AxB). Utilizando *pragma parallel omp for* Paralelizarlo de dos formas:

   a) Repartiendo entre los threads el cálculo de las filas de C. *Es decir, repartiendo el trabajo del primer for.*
   b) Repartiendo el cálculo de las columnas de cada fila de C. *Es decir, repartiendo el trabajo del segundo for.*

   Comparar los tiempos de ambas soluciones variando el número de threads.

   Ejecución: *./matrices N cantidadDeThreads*

4. El programa *traspuesta.c* calcula la transpuesta de una matriz triangular de NxN. Compilar y ejecutar para 4 threads comparándolo con el algoritmo secuencial.
   El programa tiene un problema, describir de que problema se trata. ¿Qué cláusula usaría para corregir el problema? Describir brevemente la cláusula OpenMP que resuelve el problema y las opciones que tiene. Corregir y ejecutar de nuevo comparando con los resultados anteriores.

   Ejecución: *./traspuesta N cantidadDeThreads*

5. El programa mxm.c realiza 2 multiplicaciones de matrices de NxN (D=AB y E=CB). Paralelizar utilizando secciones de forma que cada una de las multiplicaciones se realice en una sección y almacenar el código paralelo como mxmSections.c. Compilar y ejecutar sobre diferente número de threads.

   Probar con 2 threads. Luego con 4 threads ¿Se Consigue mayor speedup al incrementar la cantidad de threads? ¿Por qué?

   Ejecución: *./mxm N cantidadDeThreads*

Práctica 4

Programación con MPI

Estructura general de programa para MPI:

```
fProcesoTipoA(){
 // Función que implementa el comportamiento de los procesos de tipo A
}

fProcesoTipoB(){
 // Función que implementa el comportamiento de los procesos de tipo B
}

fProcesoTipoC(){
 // Función que implementa el comportamiento de los procesos de tipo C
}

int main(int argc, char* argv[]){
 MPI_Init(&argc, &argv);
 int id;
 int cantidadDeProcesos;

        MPI_Comm_rank(MPI_COMM_WORLD,&id);
        MPI_Comm_size(MPI_COMM_WORLD,&cantidadDeProcesos);
…
        if(id == 0)
                fProcesoTipoA();
        else if (id >= 1 && id <= 3)
                fProcesoTipoB();
        else
                fProcesoTipoC();

       MPI_Finalize();
       return(0);
}
```

NOTA: El ejemplo de programa anterior supone que habrá tres tipos de procesos con tres comportamientos diferentes. El comportamiento del proceso con id=0 será implementado en la funcion fProcesoTipoA(). El comportamiento de los procesos con id=1, 2 y 3 será implementado en la función fProcesoTipoB(). El comportamiento de los procesos restantes con id=4 a cantidadDeProcesos será implementado en la función fProcesoTipoC().

Pautas:

Compilar en Linux OpenMPI:

mpicc –o salidaEjecutable archivoFuente

Ejecutar en OpenMPI:

En una sola máquina:

mpirun –np cantidadDeProcesos ejecutable

En un cluster de máquinas:

mpirun –np cantidadDeProcesos –machinefile archivoMaquinas ejecutable

El formato de archivo de máquinas es:

maquina1 slots=cantidad de procesadores de la maquina1
maquina2 slots=cantidad de procesadores de la maquina2
…
maquinaN slots=cantidad de procesadores de la maquinaN

En todos los ejercicios implementar el algoritmo secuencial y el algoritmo paralelo. Probar el algoritmo paralelo sobre:

1) *Una máquina con 4 procesos*
2) *Dos máquinas con 2 procesos por máquina (4 procesos)*
3) *Dos máquinas con 8 procesos.*

Calcular el Speedup y la eficiencia. Realizar el análisis de escalabilidad y analizar el overhead introducido por comunicación.

1. Resolver una multiplicación de matrices de NxN y analizar los tiempos de comunicación utilizando:

   a) Operaciones punto a punto.
   b) Operaciones colectivas

   Utilizar tamaños de matrices de 512, 1024 y 2048.

2. Realizar un algoritmo paralelo que dada una matriz A de NxN obtenga el valor máximo, el valor mínimo y valor promedio de A, luego debe armar una matriz B de la siguiente forma:

   - Si el elemento $a_{i,j}$ < promedio(A) entonces $b_{i,j}$ = min(A).
   - Si el elemento $a_{i,j}$ > promedio(A) entonces $b_{i,j}$ = max(A).
   - Si el elemento $a_{i,j}$ = promedio(A) entonces $b_{i,j}$ = promedio(A).

3. Dado un vector de N elementos enteros creciente, paralelizar un algoritmo que determine cada cuanto se duplica cada número de la serie. (-1 si no se duplica)

4. Realizar un algoritmo paralelo que ordene un vector de N elementos por mezcla.

5. Dado un texto representado por un vector T de tamaño N, se debe realizar un algoritmo paralelo que obtenga la lista de palabras de T, y luego determine la cantidad de veces que aparece cada palabra en el texto quedándose sólo con las cinco palabras más frecuentes.

Práctica 5

Híbridos – Introducción a GPGPU

Pautas:

Compilar los ejercicios de acuerdo a lo visto en las clases teóricas.
Híbridos: Calcular el speedup y la eficiencia del algoritmo paralelo respecto del algoritmo secuencial.
GPU: Calcular sólo el speedup del algoritmo paralelo respecto al algoritmo secuencial ejecutado sobre una CPU.

1. Implementar un algoritmo híbrido MPI-Pthreads y uno MPI-OpenMP que resuelvan la multiplicación de matrices cuadradas de *NxN*. Obtener el tiempo de ejecución para N=512, 1024 y 2048. Ejecutar sobre 2 máquinas y 4 threads por máquina.

2. Implementar un algoritmo híbrido MPI-Pthreads y uno MPI-OpenMP que resuelvan la búsqueda del mínimo y el máximo valor en un vector de N elementos. Ejecutar sobre 2 máquinas y 4 threads por máquina.

3. Paralelizar sobre GPU un algoritmo que calcule la suma de matrices cuadradas de *NxN*. Ejecutar con 256 y 512 hilos por bloque.

4. Paralelizar sobre GPU un algoritmo que calcule el valor promedio de N elementos almacenados en un vector. Ejecutar con 256 y 512 hilos por bloque.