Computación Paralela y Distribuída
Proyecto

Prof. José Fiestas
jfiestas@utec.edu.pe
UTEC

Instrucciones para desarrollar el proyecto:

- formen grupos de 3 estudiantes
- la entrega será: el proyecto programado en C/C++, y un informe en LaTeX
- El informe debe tener capítulos correspondientes a: **Introducción** (con la descripción del problema), **Método y Desarrollo**, **Resultados y Conclusiones**
- escoja uno de los 4 ejercicios. La calificación será sobre 20
- La nota de proyecto será igual a $\mathbf{P} = \mathbf{0.5} \cdot \mathbf{D} + \mathbf{0.4} \cdot \mathbf{Pe} + \mathbf{0.1} \cdot \mathbf{Po}$, donde **D** es el desarrollo de códigos de programación, **Pe** es la presentación escrita , y **Po** es la presentación oral
- 30% de la nota **D** estará destinada a evaluar el nivel de optimización en el desarrollo del proyecto (último punto en cada propuesta)
- Entrega: viernes 11.12 por Gradescope

1 TSP

Programe en paralelo, el problema del agente viajero en paralelo utilizando el método de *branch and bound*
Se trata de un vendedor que debe minimizar el recorrido entre n ciudades. Puede empezar en cualquiera de ellas y terminar en la misma luego del recorrido. Cada ciudad debe ser visitada solo una vez. Las distancias entre ciudades son datos del problema.
Según el método de búsqueda primero en profundidad (DFS en inglés), se consideran las posibles ciudades que se pueden visitar desde una ciudad de partida. Asi sucesivamente, y calcular el camino mínimo de todas las posibilidades. Por ello, para n ciudades, obtenemos una cantidad total de caminos de (n-1)!
El algoritmo recursivo correspondiente se puede formular de la siguiente manera:

```
DFS(camino){
if (camino tiene longitud n) {
.       if (camino es el mejor camino) {
.               mejor_camino = camino
.       }
.       else: {
.               for (para todos los caminos aún no recorridos i)
.                       DFS(camino ∪ i)
.               }
.       }
}
```

En el que **camino** es una estructura que contiene las ciudades a visitar.
En **branch and bound** se verifica si el camino encontrado hasta el momento, es mayor que el mejor camino encontrado. En caso lo sea, se detiene la búsqueda por ese camino.

```
BB(camino){
if (camino tiene longitud n) {
.       if (camino es el nuevo mejor camino) {
.               mejor_camino = camino
.       }
.       else: {
.               for (para todos los caminos aún no recorridos i)
.                       if (camino ∪ i es menor que el actual mejor camino )
.                               BB(camino ∪ i)
.               }
.       }
}
```

Se muestra el algoritmo no-recursivo para su implementación en paralelo

camino={0}

```
push(camino)
while pop(camino){
if (camino tiene longitud n) {
.       if (camino es el nuevo mejor camino) {
.               mejor_camino = camino
.       }
.       else: {
.               for (para todos los caminos aún no recorridos i)
.                       if (camino ∪ i es menor que el actual mejor camino )
.                               push(camino ∪ i)
.               }
.     }
}
```

Para su implementación se utiliza una pila (stack), que se inicializa con la ciudad **0**. En cada iteración se retira la ciudad del tope de la pila. Si el camino tiene longitud n se evalua, de lo contrario se añaden ciudades si la longitud del camino no es mayor que el del mejor actual. El programa termina cuando la pila está vacía.

Se propone lo siguiente:

a) Programar TSP en C++, escoger el paradigma apropiado de paralelismo, e implementarlo

b) Registrar el desarrollo del código de manera ordenada en por lo menos 3 pasos (versiones beta), hasta llegar al código final

c) Estimar la complejidad del código secuencial y en paralelo, y compararla con métricas de tiempo presentadas en gráficas adecuadas

d) Medir la velocidad del algoritmo y representarla en gráficas. Medir la escalabilidad del software

e) Optimizar el software con un desarrollo orientado al paralelismo, y presentarlo en las conclusiones del paper

2 Expresiones regulares

Expresiones regulares son una forma de representar un conjunto de cadenas de caracteres que satisfacen ciertas reglas de construcción (gramáticas). E.g. dado el alfabeto $\Sigma = \{0, 1\}$, la expresión regular $01(01)^*$ acepta cadenas como {01,0101,010101,...}. Expresiones regulares se usan frecuentemente en algoritmos de manipulación de strings, de análisis lexicográfico y sintáctico, entre otros. Un ejemplo de ello es NetKat, un lenguaje de programación de redes [1]

Un software de reconocimiento de expresiones regulares es de mucha utilidad, especialmente si es escalable. PAREM [2] es un ejemplo de un sotware de expresiones regulares rápido y escalable, con un speedup lineal con respecto al número de procesos. Este algoritmo particiona la cadena (input), para luego combinar los resultados parciales obtenidos (ver paper).

Se propone lo siguiente:
a) Programar PAREM en C++, escoger el paradigma apropiado de paralelismo, e implementarlo
b) Registrar el desarrollo del código de manera ordenada en por lo menos 3 pasos (versiones beta), hasta llegar al código final
c) Estimar la complejidad del código secuencial y en paralelo, y compararla con métricas de tiempo presentadas en gráficas adecuadas
d) Medir la velocidad (en FLOPs) del algoritmo y representarla en gráficas. Medir la escalabilidad del software
e) Optimizar el software con un desarrollo orientado al paralelismo, y presentarlo en las conclusiones del paper

Bibliografía:
[1] www.cs.princeton.edu/∼dpw/papers/frenetic-netkat.pdf
[2] arxiv.org/pdf/1412.1741.pdf

3 RngStreams: generador random

Uno de los principales objetivos de la generación de números random en paralelo, es lograr la independencia de procesos al generar éstos. Esto colind con el paralelismo ideal, pero es uno de los fundamentos principales de modelos estocásticos.

Los distintos métodos incluyen:
a) **Random spacing**: donde los procesos se inicializan en forma random a través de distintos seeds (E.g. Mersenne Twister (MT) generator)
b) **Sequence splitting**: dividiendo una secuencia en bloques que no se superponen
c) **Cycle division**: donde el período de un generador se divide en segmentos en forma determinística (Combined Multiple recursive generator (CMRG) with cycle division: RngStreams package)
d) **Parametrización**: los parámetetros de un generador se modifican para producir distintos streams (Multiplicative lagged Fibonacci generator with parametrization: SPRNG package)

El método MRG garantiza que los streams no coincidan. Este puede usarse en forma combinada (CMRG), con mayores ventajas en la generación de random. Son muy estables y estadísticamente seguros. El paquete RngStreams puede ser encontrado aqui [1]
Se propone lo siguiente:
a) Estudiar y probar el algoritmo RngSTreams en paralelo [2], utilizando MPI u OMP
b) Implemente el algoritmo RngStreams en un código de integración por Monte Carlo c) Registrar el desarrollo del código de manera ordenada en por lo menos 3 pasos (versiones beta), hasta llegar al código final
d) Estimar la complejidad del código secuencial de integración y en paralelo, y compararla con métricas de tiempo presentadas en gráficas adecuadas
d) Medir la velocidad (en FLOPs) del algoritmo y representarla en gráficas. Medir la escalabilidad del software
e) Optimizar el software con un desarrollo orientado al paralelismo, y presentarlo en las conclusiones del paper

Bibliografía
[1] http://www.iro.umontreal.ca/ lecuyer/myftp/streams00/
[2] https://arxiv.org/pdf/1403.7645.pdf

4 Colisión galáctica

Simulaciones de colisión geláctica son unas de las más importantes para poder predecir teóricamente el comportamiento de dos galaxias en cercanía y con probabilidad de colisión en el futuro.

El modelo requiere conjuntos de objetos (estrellas) de una cantidad muy grande, billones de estrellas en la realidad, que exigen una simulación lo más realistica posible, pero cuyos resultados sean accesibles en un tiempo mesurado.

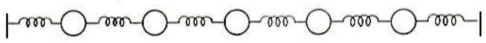
Esta propuesta se basa en el trabajo hecho por el grupo Astrofísica de la Universidad de Heidelberg, Alemania. Para ello, se cuenta con el software PhiGPU [1].

Se propone lo siguiente:
a) Estudiar y probar el algoritmo phiGPU en paralelo [2], utilizando MPI
b) Modifique las condiciones iniciales adecuadas para simular la colisión de dos galaxias y ejecute el código
c) Registrar el desarrollo de las simulaciones de manera ordenada en por lo menos 3 pasos (versiones beta), hasta llegar al código final
d) Estimar la complejidad del código secuencial y en paralelo, y compararla con métricas de tiempo presentadas en gráficas adecuadas
d) Medir la velocidad (en FLOPs) del algoritmo y representarla en gráficas. Medir la escalabilidad del software
e) Optimizar el software con un desarrollo orientado al paralelismo, y presentarlo en las conclusiones del paper

Bibliografía
[1] phiGPU code
[2] Paper2011

5 Oszilador Armónico acoplado

Analice la dinámica de una cadena que representa un oscilador armónico acoplado de N partículas (átomos). Cada átomo está acoplado a su átomo vecino por una constante elástica acoplada D.



La ecuación clásica de movimiento es:
$mx.. = -D(x_i - x_{i+1}) - D(x_i - x_{i-1}) = -D(2x_i - x_{i+1} - x_{i-1})$

Donde x_i representa la desviación de la partícula i de su posición de reposo. Los extremos de la cadena estan fijos a $x_0 = x_n =$ constante, con las relaciones $\sqrt{m}x_i(t) = v_i exp(iwt)$

El cálculo de las vibraciones del sistema, se reduce al cálculo de los valores propios y vectores propios de la matriz tridiagonal
$M_{ij} = \frac{D}{\sqrt{m_i m_j}}(2\delta_{ij} - \delta_{i,j+1} - \delta_{i,j-1})$

1. Analice el caso de un defecto en la cadena, en el que m_i = m = constante, para $n \neq n_0$, y m_n = 100 m para $n = n_0$
Escoja valores de tiempo y unidades de longitud adecuadas, para que m = D = 1
Escriba un programa en paralelo que resuelva la matriz y calcule los valores y vectores propios para N=99, n_0=60 y n_0=30 con ayuda de la librería tqli de Numerical Recipees [1]

2. Escoja condiciones iniciales $x_i(0)$=1 (i=n_0) y $x_i(0)$=0 ($i \neq n_0$) y grafique $x_i(t)(i = n_0)$ para un intervalo adecuado de tiempo.
Nota: las condiciones iniciales están representadas como una superposición de las vibraciones individuales.
$y_i = \sum_{n=1}^{N-1} a_n v_n e^{iw_i t} + b_n v_n e^{-iw_i t}$
O en valores reales
$y_i = \sum_{n=1}^{N-1} a_n v_n cos(w_i t) + b_n v_n sin(w_i t)$
Se propone lo siguiente:
a) Desarrolle puntos 1 y 2 de la descripción. Elevar la dimensión de la cadena lo suficiente para obtener mediciones significativas de tiempos
b) Registrar el desarrollo de las simulaciones de manera ordenada en por lo menos 3 pasos (versiones beta), hasta llegar al código final
c) Estimar la complejidad del código secuencial y en paralelo, y compararla con métricas de tiempo presentadas en gráficas adecuadas
d) Medir la velocidad (en FLOPs) del algoritmo y representarla en gráficas. Medir la escalabilidad del software
e) Optimizar el software con un desarrollo orientado al paralelismo, y presentarlo en las conclusiones del paper

Bibliografía

[1] https://www.cec.uchile.cl/cinetica/pcordero/MClibros/NumericalRecipesinC.pdf