I - S - 2021 - RRF - Programación Paralela y Concurrente - 002

La segunda entrega del Proyecto 2 consiste en mejorar el rendimiento de la primera entrega agregando distribución mediante la tecnología `MPI`.

El se realiza en grupos de máximo tres personas. Las entregas tardías se tratarán como se estipula en la carta al estudiante.

Deben entregar su solución en el repositorio privado que para ese efecto ya creó cada grupo, dentro de una carpeta exclusiva para este proyecto, dado que el repositorio albergará los dos proyectos del curso. Deben asegurarse de que el docente a cargo tenga acceso a dicho repositorio. Todos los miembros del grupo deben demostrar un nivel similar de participación en el desarrollo de la solución. Para esto se les recomienda hacer `commits` frecuentes con comentarios breves pero descriptivos de lo que representa cada `commmit`.

Revisado en el repo. Mayor variedad commits

En esta segunda entrega se hace énfasis en el uso de un mecanismo de programación paralela en un ambiente de memoria distribuída.

La descripción del problema así como los requerimientos de la solución se mantienen igual que en la primera entrega excepto por la modificación que se describe a continuación y es de caracter opcional.

Diseño de la aplicación distribuida [20%] ✓

Cree un documento en la carpeta `design/` del Proyecto02 con un archivo en formato `markdown` llamado `design_analysis.md` con una descripción de no más de 500 palabras que incluya al menos los siguientes puntos:

1. Descripción del tipo descomposición y mapeo a utilizar para la parte distribuida del programa (mediante `mpi`).
2. Descripción del tipo descomposición y mapeo a utilizar para la parte concurrente del programa (mediante `OpenMP`).
3. Explicación de al menos una ventaja y una desventaja que encuentra para cada uno de los puntos anteriores, es decir, su propuesta de descomposición y mapeo para OpenMP y MPI.
4. Si utilizan separación de asuntos, explicar qué hace cada tipo de proceso y en qué nodo se ejecuta.
5. Describa cómo los procesos van a manejar la entrada y la salida.

Implementación distribuida [50%]

- Solucionar espera activa.

Sobre lasolución concurrente mediante OpenMP, realicen una paralelización que incremente el desempeño usando la tecnología MPI.

Antes de continuar, meda el rendimiento de la versión **concurrente OpenMP** con el caso de prueba `job002.txt` y realizar mediciones de rendimiento en un nodo esclavo del clúster Arenal y anotar el resultado en una hoja de cálculo.

Luego, Implemente la paralelización con MPI tal y como lo planteó en el apartado anterior **Diseño de la aplicación distribuida**. La cantidad de nodos esclavos a utilizar debe ser la cantidad de nodos disponibles en el cluster de Arenal y la cantidad de CPUs a utilizar debe poder ser controlada por argumento de línea de comandos. En caso de omitirse se debe suponer la cantidad de CPU disponibles en el equipo donde corre el programa.

Asegúrese de que la solución pasa las pruebas de corrección. La solución debe pasar los casos de prueba, y no tener malas prácticas como espera activa, condiciones de carrera, bloqueos mutuos, inanición, serialización

innecesaria, o ineficiencia. Se recomienda usar herramientas como `tsan` para validar el correcto funcionamiento del programa y `cpplint` para validad el apego a una convención de estilo del código fuente.

Comparación de rendimiento [25%]

Ahora, realize las siguientes mediciones de rendimiento de sus programas utilizando el caso de prueba `job002.txt`. .

1. Mida el rendimiento de la versión **distribuida (MPI)** en todos los nodos del cluster usando tantos procesos por nodo como núcleos haya en dicho nodo. Por ejemplo en el cluster de Arenal serían veinticuatro procesos (8 por nodo) y un hilo en cada proceso.
2. Mida el rendimiento de la versión **híbrida (OpenMP+MPI)** en todos los nodos del cluster usando un proceso por nodo, y tantos hilos como núcleos haya en dicho nodo. Por ejemplo en el cluster de Arenal serían tres procesos y ocho hilos en cada proceso.

Calcular el incremento de velocidad (speedup) y eficiencia utilizando el caso de prueba `job002.txt`de las siguientes versiones del programa. Cree un gráfico que incluya en el eje-x tres mediciones: (a) **Concurrente OpenMP** en un solo nodo, (b) **distribuida MPI** 24 procesos, y (c) **híbrida OpenMP+MPI** tres procesos y ocho hilos. En el eje-y primario muestre el incremento de velocidad, y en el eje-y secundario la eficiencia. Incruste su gráfico en una sección "Análisis de rendimiento" de su `README.md`.

Comente brevemente (500 palabras máximo) sí para este problema las decisiones sobre la descomposición y mapeo que propuso le permitieron obtener los resultados que esperaba. Indique de qué manera podría mejorar la eficiencia de su programa o solución.

Estructura del proyecto [5%] ✓

Buen uso del repositorio de control de versiones. Debe tener *commits* de cada integrante del equipo. Debe haber una adecuada distribución de la carga de trabajo, y no desequilibrios como que “un miembro del equipo sólo documentó el código”. Cada *commit* debe tener un cambio con un significado, un mensaje significativo, y “no romper el build”.

Las clases, métodos, y tipos de datos, deben estar documentados con Doxygen. La salida de Doxygen debe estar en una carpeta `doc/` y ésta NO se debe agregar a control de versiones. Documentar el código no trivial en los cuerpos de las subrutinas. Debe mantenerse la estructura de archivos y directorios característica de un proyecto de Unix, resumido en la siguiente tabla.

Recurso	Descripción	Versionado
`bin/`	Ejecutables	No
`build/`	Código objeto (.o) temporal	No
`design/`	Diseño de la solución en redes de Petri o pseudocódigo o ambos	Sí
`doc/`	Documentación generada por doxygen	No
`src/`	Código fuente (.hpp y .cpp)	Sí
`test/`	Casos de prueba **comprimidos**	Sí
`Makefile`	Compila, prueba, genera documentación, limpia	Sí
`Doxyfile`	Configura Doxygen para extraer documentación del código fuente	Sí
	Presenta el proyecto,	

`README.md`	incluye el manual de usuario	Sí
`.gitignore`	Ignora los recursos que NO deben estar en control de versiones (columna Versionado)	Sí

Debe crear un archivo `README.md` en la raíz de la solución que indique el propósito de la misma. Agregar una sección "Compilar" (en inglés, "Build") al `README.md` que explique cómo compilar y correr su código. Agregar en una sección "Manual de usuario" cómo ejecutar la aplicación, describir los parámetros, y describir la salida esperada. Conviene agregar salidas textuales o capturas de pantalla. Incluir una sección de diseño en el `README.md` con el modelo estático (diagrama de clases) y, a modo opcional, el modelo dinámico (diagrama de secuencia o actividad) de su solución.

[Aspectos transversales]

El peso en la evaluación de cada componente del avance se encuentra en los títulos anteriores. En cada uno de ellos se revisa de forma transversal:

1. La corrección de la solución. Por ejemplo: comparar la salida del programa contra los casos de prueba.
2. La completitud de la solución. Es decir, si realiza todo lo solicitado.
3. La calidad de la solución ejecutable. Por ejemplo, no generar condiciones de carrera, espera activa, bloqueos mutuos, inanición (puede usar herramientas como `helgrind`, `tsan`). Tampoco accesos inválidos ni fugas de memoria (`memcheck`, `asan`, `msan`, `ubsan`).
4. La calidad del código fuente, al aplicar las buenas prácticas de

programación. Por ejemplo, modularizar (dividir) subrutinas o clases largas, y modularizar los archivos fuente (varios `.hpp` y `.cpp`). Apego a una convención de estilos (`cpplint`). Uso identificadores significativos. Inicialización de todas las variables. Reutilización de código

Casos de prueba

test_set_1.zip

test_set_2.tar.gz