Informe Trabajo Final

OMP / MPI - Programación Concurrente

Docente: Maximiliano A. Eschoyez.

Alumnos: BOZZANO, Felipe Gabriel.
OLMOS BOSSA, Francisco Augusto.

Institución: Universidad Blas Pascal.

Carrera: Ingeniería Informática.

Asignatura: Programación Concurrente.

Fecha entrega: 26/11/2021

BOZZANO, Felipe
OLMOS, Francisco

Problemática

Resumen

Realizaremos un programa de software implementando diferentes métodos de integración utilizando programación paralela o concurrente, por medio de las API de OMP para el modelo de hilos con uso de memoria compartida y MPI (Interfaz de paso de mensajes) para el modelo de procesos sin memoria compartida. Además, realizamos una serie de mediciones de tiempo de ejecución para comparar el programa secuencial, el comportamiento del programa paralelizado con múltiples hilos y con múltiples procesos.

Consigna

En este trabajo se deben desarrollar dos versiones de un programa, tanto para su implementación con la biblioteca OpenMP y con MPI (son 4 programas en total), para realizar la integración numérica de diferentes funciones.

El objetivo es dividir el espacio de integración entre todos los procesos/hilos que se vayan a ejecutar. De esta forma, la cantidad de procesos/hilos dependerá de los recursos disponibles o la configuración de inicio, y cada proceso buscará sólo en el rango que le corresponde.

La primer versión del programa debe ejecutar cada método de integración en un hilo diferente para OpenMP o en un proceso para MPI.

La segunda versión debe dividir la tarea de integración de cada método en n hilos OpenMP o procesos MPI y, además, hacer que se ejecuten los cuatro métodos de integración en paralelo. Es decir, si n = 4 se tienen que ejecutar los 4 métodos con 4 hilos/procesos cada uno, totalizando 16 hilos/procesos en la CPU.

Los métodos de integración a implementar son:

1. Regla del Rectángulo,
2. Regla del Punto Medio,
3. Regla del Trapecio,
4. Simpson 1/3.

Solución

Secuencial

Para esta implementación no paralelizamos la ejecución de los métodos de integración numérica, sino que el proceso principal los ejecuta uno a uno.

OpenMP V1

En esta versión utilizamos la API de hilos paralelos OMP para construir una solución en la cual cada método de integración es ejecutado por un hilo distinto. Utilizamos algunas directivas de OMP como **#pragma omp parallel** para indicar que a partir de ese bloque de código va a comenzar la concurrencia y paralelismo entre los hilos. A su vez hicimos de uso de **#pragma omp sections** para generar una serie de secciones, cada una manejada para un hilo, las cuales van a alojar un método numérico distinto.

OpenMP V2

En esta versión utilizamos la API de hilos paralelos OMP para construir una solución en la cual cada método de integración es ejecutado por una serie de hilos distintos. Utilizamos las mismas directivas que para la versión 1, pero como la problemática es que cada método de integración sea resuelto por n hilos, agregamos las siguientes directivas: **omp_set_nested(1)** para habilitar la anidación de paralelismo. **#pragma omp for schedule reduction** la cual divide un for en porciones (chunks) que serán ejecutadas por un hilo distinto, y al finalizar cada uno sumara su resultado en una misma variable compartida.

MPI V1

En esta versión utilizamos la API MPI (Interfaz de paso de mensajes) para construir una solución en la cual cada método de integración es ejecutado por un proceso distinto. Utilizamos una serie de funciones de la API como ser **MPI_Init** para comenzar el contexto de ejecución de MPI, y generamos una comunicación punto a punto entre procesos parecida a la de la cola de mensajes, en donde un proceso root es el encargado de recibir mensajes (resultados de los métodos de integración) de los demás procesos por medio de **MPI_Recv** (bloqueate) y los demás procesos envían el resultado del método de integración con **MPI_Send** (bloqueante). Para liberar el contexto de MPI utilizamos **MPI_Finalize**.

MPI V2

En esta versión utilizamos la API de hilos paralelos OMP para construir una solución en la cual cada método de integración es ejecutado por una serie de procesos distintos. Lanzamos la cantidad de proceso indicados y los dividimos en clusters, cada cual se encarga de resolver un método numérico distinto. Para ello utilizamos **MPI_Comm_split** en donde cada proceso adquiere un identificador distinto dentro del cluster. Utilizamos también **MPI_Barrier** para bloquear la ejecución de todos los procesos hasta que todos lleguen hasta esta instrucción, esto lo hacemos para poder tomar exactamente el tiempo de ejecución.

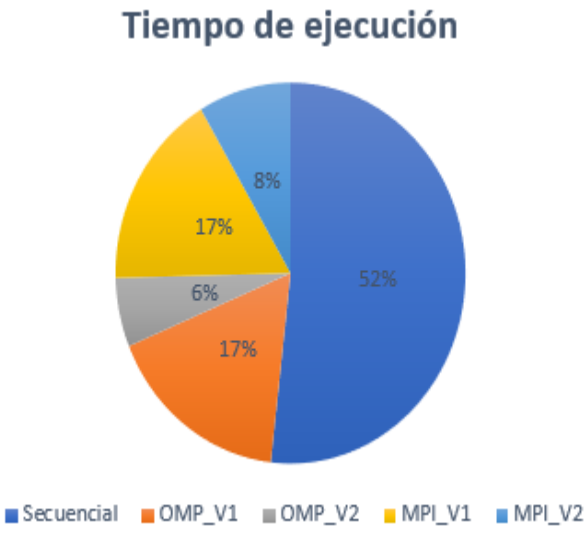
Mediciones

Datos

- Función: 2x^2 + 3x - 1
- Rango de integración: [0, 15]
- Cantidad de intervalos: 90.000.000
- Cantidad de muestras: 5

Resultados

Secuencial – Total	OMP_V1 - Total	OMP_V2 - Total	MPI_V1 - Total	MPI_V2 - Total
4.5729	1.4854	0.5138	1.4762	0.8833
4.5263	1.4940	0.5164	1.4775	0.7577
4.5732	1.4814	0.4934	1.4811	0.6966
4.5360	1.4848	0.5291	1.4853	0.7475
4.5424	1.4899	0.5211	1.4796	0.6683



Conclusión

Observando los gráficos, vemos claramente que el proceso secuencial lleva un tiempo de ejecución mucho más grande que los procesos concurrentes. Esto se hace notorio cuando utilizamos una cantidad de iteraciones muy grande (90 millones en este caso), mientras que a medida esa cantidad disminuye, la diferencia de tiempo se achica a gran escala.

Respecto a el lanzamiento de 1 hilo/proceso por método de integración, vemos que casi no hay diferencia entre ambos. Es interesante mencionar que en esta versión, la implementación con procesos requiere de una comunicación entre los 4 procesos que hacen el esfuerzo matemático, y el proceso padre que espera recibir y mostrar por pantalla, mientras que la versión de los hilos no tiene este tipo de comunicación, sino que el padre espera que los hilos finalicen para poder cerrar el programa. Aún así, vemos que ese tiempo de comunicación es despreciable respecto al tiempo de cálculo computacional.

Donde si encontramos diferencia, es en las versiones de multihilos/multiprocesos, donde la concurrencia de hilos muestra una mayor velocidad que la de procesos. Ésto puede deberse a que los hilos comparten la memoria en su ejecución, mientras que los procesos no lo hacen, por lo que MPI tiene que resolver como manejar las variables resultado de cada método numérico.