Algoritmos Paralelos

Profra: Luz Gasca Soto Febrero 1, 2019
Ayudantes: Antonio Alvarez / Jorge García

Practica 1

1 Introducción a OpenMP

En este PDF se mencionarán las cosas más relevantes que los alumnos deben tener en cuenta para poder realizar las prácticas de este curso,para aprender más de OpenMP, se recomiendan visitar las siguientes dos páginas:

1. Puedes visitar el tutorial en linea que se encuentra en la pagina: Livermore computing center.
2. En el sitio oficial de OpenMP.

1.1 OpenMP

OpenMP (Open specifications for MultiProcessing) es una API (Application Program Interface) estandarizada que soporta paralelismo basado en threads. Provee un modelo explícito de programación para controlar la creación, comunicación y sincronización de múltiples threads. OpenMP usa el modelo fork-join de ejecución paralela:

1. Cada programa en OpemMP empieza con un main thread, el cual es ejecutado de manera secuencial, hasta llegar a la sección paralela.
2. El main thread es el encargado de crear a los otros threads que serán ocupadas.
3. Las declaraciones en el programa encerradas por la región paralela son ejecutadas entonces, simultáneamente entre el equipo de threads.
4. Cuando el equipo de threads completa las declaraciones de la región paralela, se sincronizan y terminan, dejando solo el thread principal.

La API de OpenMP queda comprendida en tres componentes principales:

1. Directivas de compilador: especifican paralelismo en el codigo fuente de C.
2. Rutinas en tiempo de ejecución: Proveen acceso a los parámetros en tiempo de ejecución.
3. Variables de entorno: dirigen como debe comportarse el sistema.

2 Directivas de compilador

Virtualmente todo el paralelismo en OpenMP se especifica mediante el uso de directivas de compilador las cuales son embebidas dentro de código de C (o C++ o Fortran). Estas directivas siguen las convenciones del compilador del lenguaje original C. Existen cuatro tipos diferentes de directivas:

1. Constructor paralelo.
2. Constructor de trabajo compartido.
3. Constructor combinado de trabajo compartido paralelo.
4. Directivas de sincronización.

2.1 Constructor paralelo

Una región paralela es un bloque de código que será ejecutada por múltiples threads. Este es el constructor fundamental paralelo de OpenMP. La sintaxis de este constructor es como sigue:

```
#pragma omp parallel [clause ...]
        if (scalar\_expression)
        private (list)
        shared (list)
        default (shared | none)
        firstprivate (list)
        reduction (operator: list)
        copyin (list)
structured\_block
```

Cuando un thread llega a la directiva PARALLEL, crea un grupo de threads y se devuelve el main thread de ese grupo. El main es miembro de ese grupo y tiene el thread número 0 dentro del grupo. Comenzando desde el inicio de la región en paralelo, el código es duplicado y todos los threads van a ejecutar ese código. Hay una barrera implícita al final de una sección paralela. Solo el main thread continúa su ejecución pasado este punto. El número de threads en una región paralela se determina por los siguientes factores en orden de precedencia:

1. Uso de la función omp_set_num_threads().
2. Cambiando la variable de entorno OMP_NUM_THREADS.
3. Implementación de defecto.

2.2 Constructor de trabajo compartido

Un constructor de trabajo compartido divide la ejecución del código encerrado en la región paralela entre los threads que se encuentran. Este tipo de constructores no lanza nuevos threads y no se tiene una barrera implícita al entrar a un constructor de trabajo compartido, aun así hay una barrera implícita al final del constructor. Existen tres constructores de trabajo compartido cuya sintaxis es como sigue:

```
#pragma omp for [clause ...]
        schedule (type [,chunk])
        ordered
        private (list)
        firstprivate (list)
        lastprivate (list)
        shared (list)
        reduction (operator: list)
        nowait
 for_loop
```

En OpenMP, las iteraciones son asignadas a los threads en rangos continuos llamadas porciones, controlando como es que estas porciones se asignan a los threads y el número de iteraciones por porción, un esquema de calendarización hace el balance del trabajo entre los threads. La clausula schedule describe cómo las iteraciones del loop son divididas entre los threads en el grupo. Existen 4 disponibles, aquí la descripción de dos de ellas: 1. static: Las iteraciones del loop son divididas en porciones o piezas del tamaño de

la porción y luego estáticamente asignadas a los threads. 2. dynamic: Las iteraciones del loop son divididas en porciones o piezas del tamaño de la porción y luego estáticamente asignadas a los threads. Cuando un thread finaliza una porción, se le asigna dinámicamente otra. El tamaño de porción por defecto es 1.

2.3 Constructor combinado de trabajo compartido paralelo

Un constructor combinado de trabajo compartido paralelo es un atajo para especificar una región paralela que solo contiene un constructor de trabajo compartido. La semántica de esta directiva es idéntica al caso de explícitamente especificar una directiva parallel seguida de un solo constructor de trabajo compartido.

```
#pragma omp parallel sections [clause ...]
        default (shared | none)
        shared (list)
        private (list)
        firstprivate (list)
        lastprivate (list)
        reduction (operator: list)
        copyin (list)
        ordered
structured_block
```

3 Directivas de sincronización

Existen varios constructores disponibles para coordinar el trabajo de múltiples threads. A continuación se en-listarán las más importantes para este curso.

1. master: Especifica una región que se ejecutará únicamente por el main thread del grupo. Todos los otros threads saltaran esta sección de código.

```
#pragma omp master
structured_block
```

2. critical: Especifica una región de código que debe ejecutarse por solo un thread a la vez.

```
#pragma omp critical
structured_block
```

3. barrier: Sincroniza todos los threads en el grupo. Cuando se llega a una directiva de barrera un thread debe esperar en ese punto hasta que todos los demás threads hayan llegado a la barrera. Todos los threads entonces reanudan su ejecución en paralelo.

```
#pragma omp barrier
```

4. atomic: Especifica que una localidad de memoria debe actualizarse atómicamente, en lugar de dejar múltiples threads intentando escribir en ella. En esencia, esta directiva provee una sección crítica miniatura.

```
#pragma omp atomic
statement_expression
```

5. ordered: Especifica que las iteraciones de un loop deben ejecutarse en el mismo orden como si fueran ejecutadas en un procesador secuencial.

```
#pragma omp ordered
structured_block
```

4 Cláusulas de alcance de variables

Una consideración importante para la programación de OpenMP es comprender el alcance de datos o scope. Dado que OpenMP está basado en la compartición de memoria la mayoría de variables son compartidas por defecto. En particular, todas las variables visibles cuando se entra a la región paralela son globales (compartidas). Las variables privadas incluyen índices de loops y variables de pila creadas en a región paralela. En adición las cláusulas de alcance pueden utilizarse para definir explícitamente cómo debe ser el alcance.Unos ejemplos de esto serían:

1. shared: Declara uno o más elementos que se comparten por las tareas generadas por el constructor de la región paralela.

2. private: Declara uno o más elementos privados.

3. firstprivate: Declara uno o más elementos privados, inicializandólos cada uno en su correspondiente valor del elemento original en el momento que se encuentra en el constructor.

4. nowait: Evita la sincronización implícita de los threads al terminar el bloque de una directiva.

5 Rutinas en tiempo de ejecución

OpenMP define una API para llamadas que realizan una variedad de funciones:

1. Preguntar el número de threads/procesadores, colocar el número de threads a usar.

2. Rutinas de propósito general bloqueantes (semáforos).

3. El conjunto de funciones de entorno: paralelismo anidado, ajuste dinámico de threads.

Es necesario incluir la biblioteca **omp.h** para poder usar estas funciones. Las funciones más importantes que ocuparemos serán las siguientes:

1. Asigna el numero de threads que van a utilizarse en la siguiente región paralela:

   ```
   void omp_set_num_threads(int num threads)
   ```

2. Regresa el numero de threads en el grupo que actualmente están ejecutándose en la región paralela desde la llamada a función:

   ```
   int omp_get_num_threads():
   ```

3. Regresa el maximo valor que puede devolver omp_get_thread_num():

   ```
   int omp_get_max_threads():
   ```

4. Regresa el numero de un thread, dentro del grupo, que hace esta llamada:

   ```
   int omp_get_thread_num():
   ```

5. Regresa el numero de procesadores disponibles para el programa.

   ```
   int omp_get_num_procs():
   ```

6 Ejercicios

Dado el código fuente y el PDF de métricas, (que están en la página del curso), el alumno deberá realizar un experimento practico que consiste en realizar varias ejecuciones del programa con diferentes parámetros, tomando nota del tiempo de ejecución reportado, y utilizar esa información para calcular aproximaciones de las métricas vistas en clase, con las cuales deberán llenar la siguiente tabla:

Procesos	Tiempo de ejecución T(p)	Speedup S(p)	Eficiencia E(p)	Fracción serial F(p)
1				
2				
4				
6				
8				
10				
20				
50				
100				

Observaciones:

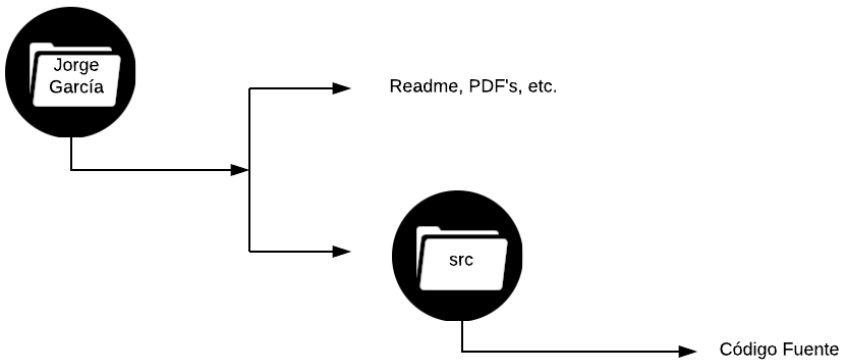
1. Para compilar el programa: gcc Practica01.c -fopenmp
2. Una vez compilado,ejecuta el código con el número de threads que deseas ocupar, por ejemplo:./a.out 4
3. Para calcular el tiempo del algoritmo secuencial, utilicen el tiempo obtenido por un proceso.
4. Como se menciona, las métricas obtenidas serán solo aproximaciones, esto debido a que se calculan a partir del tiempo de ejecución en un experimento práctico, y no con base en la complejidad del algoritmo.
5. A veces una ejecución en particular puede tomar más tiempo de lo normal debido a procesos de manejo del sistema operativo u otras circunstancias, por lo cual pude ser buena idea ejecutar varias veces con el mismo número de hilos, para así darse una mejor idea de cuánto es el tiempo "normal" y cuando son "anomalías".

Además de entregar la tabla en un archivo PDF, deberán crear tres programas en c que calculé el speedup, la eficiencia y la fracción serial, los

cuales deben recibir desde terminal los parámetros necesarios para dichos cálculos.

7 Entregas

Para la entrega de prácticas deberán crear una carpeta con su nombre y apellido en el cual guardarán los archivos readme (especificaciones sobre su programa) o PDF's (si lo requiere la practica) y una sub-carpeta llamada src el cual tendrá todos los códigos fuente.



Esta carpeta debe ser comprimida en zip y ser enviada al correo:

jorgel_garciaf@ciencias.unam.mx

con asunto **[AParalelos]PracticaN**, donde N es el número de la practica, en el cuerpo del correo deberá estar el nombre y el número de cuenta del alumno.

La fecha de entrega para la practica 1 es para el Miércoles 6 de febrero del 2019, antes de las 23:59.
No se recibirán prácticas pasada la fecha de entrega.
Si sus códigos no compilan, en automático tendrán 0 en la práctica.
Si se descubre que alguien copio en la practica, todos los involucrados en automático reprobarán el laboratorio.