September 10, 2024

ALN: Clase 7 - HPC (parte 2)

Ernesto Dufrechou
Pablo Ezzatti[1]

Instituto de Computación, Universidad de la República, Montevideo, Uruguay,

Agenda

Medidas de Performance

Las medidas de rendimiento son fundamentales en la evaluación y optimización de algoritmos paralelos. Su objetivo principal es proporcionar una estimación precisa del desempeño y permitir la comparación con algoritmos secuenciales. Factores clave incluyen:

- Tiempo de ejecución
- Utilización de los recursos disponibles

El tiempo de ejecución puede verse afectado por el almacenamiento y transmisión de datos entre procesos.

Tiempo de Ejecución y Utilización de Recursos

La utilización eficiente de los recursos y la capacidad para resolver problemas complejos son cruciales. El tiempo total de ejecución es una medida fundamental del rendimiento.

- Tiempo en estado ocioso puede afectar el rendimiento general.
- Minimizar el tiempo en estado ocioso es clave para el diseño de algoritmos paralelos.

Speed-Up

El *Speed-Up* mide la mejora de rendimiento al aumentar el número de procesadores. Existen dos tipos:

- **Speed-Up absoluto**
- **Speed-Up algorítmico**

Speed-Up Absoluto

Definition (Speed-Up absoluto)

Se define como:

$$S_N = \frac{T_0}{T_N} \tag{1}$$

- T_0: Tiempo de ejecución del mejor algoritmo serial.
- T_N: Tiempo de ejecución del algoritmo paralelo con N procesadores.

Speed-Up Algorítmico

Definition (Speed-Up algorítmico)

Se define como:

$$S_N = \frac{T_1}{T_N} \tag{2}$$

- T_1: Tiempo serial.
- T_N: Tiempo paralelo de ejecución.

Eficiencia

Definition (Eficiencia)

La eficiencia se define como:

$$E_N = \frac{T_1}{N \cdot T_N} \tag{3}$$

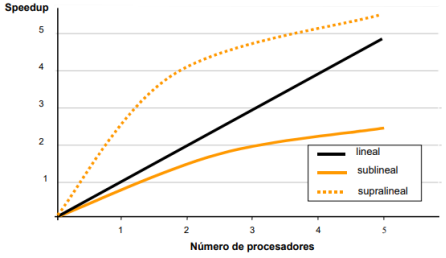
También se puede expresar como $E_N = \frac{S_N}{N}$. Una eficiencia cercana a uno indica una utilización muy efectiva de los recursos paralelos.

Speed-Up Ideal y Factores que Afectan el Desempeño

La situación ideal es un Speed-Up lineal. Sin embargo, varios factores pueden impedir un crecimiento lineal del Speed-Up:

- Demoras por comunicaciones
- Overhead en sincronización y intercambio de datos
- Tareas no paralelizables y cuellos de botella en el hardware

En algunos casos, es posible lograr un speedup superlineal ($T_1/T_N > N$) (p. ej. debido a la mejora en el acceso a memorias más rápidas).



Agenda

Ley de Amdahl - 1967

La ley de Amdahl indica que existe un límite en la mejora del rendimiento de un programa al agregar más procesadores. Se enuncia así:

> *"La parte serial de un programa determina una cota inferior para el tiempo de ejecución, aún cuando se utilicen al máximo técnicas de paralelismo."*

Esta ley destaca que la parte secuencial del programa actúa como un cuello de botella, limitando la mejora del rendimiento. La razón para agregar más procesadores es resolver problemas más grandes o complejos, no solo para acelerar problemas de tamaño fijo.

Escalabilidad

La escalabilidad se refiere a la capacidad de un programa paralelo de mejorar su rendimiento al aumentar el número de procesadores. Es una característica clave en algoritmos paralelos y distribuidos.

- **Escalabilidad fuerte**: Se refiere a obtener un speed-up cercano a lineal para un tamaño de entrada dado. Ejemplo: "El programa escala hasta 200 procesadores para un tamaño de entrada determinado."
- **Escalabilidad débil**: Se refiere al aumento del tamaño de entrada y procesadores de manera que cada procesador mantenga aproximadamente la misma carga de trabajo. El tiempo de ejecución se mantiene aproximadamente constante con el aumento proporcional de procesadores.

Utilización de Recursos Disponibles

La utilización de recursos mide el porcentaje de tiempo durante el cual un procesador está en uso durante la ejecución de una aplicación paralela. Se define como:

$$\text{USO} = \frac{\text{Tiempo ocupado}}{\text{Tiempo ocioso} + \text{Tiempo ocupado}}$$

Un objetivo importante es mantener valores equitativos de utilización entre todos los procesadores para asegurar un uso eficiente de los recursos y evitar desequilibrios significativos en la carga de trabajo.

Granularidad

La “granularidad” es un factor crucial que influye en la eficiencia de las aplicaciones paralelas. Refleja la cantidad de trabajo realizada por cada nodo o procesador.

- Granularidad fina: operaciones muy pequeñas.
- Granularidad gruesa: procesos completos.

Aumentar la granularidad puede disminuir el sobrecosto de control y comunicación, pero también puede reducir el grado de paralelismo disponible. El objetivo es equilibrar el sobrecosto de sincronización y comunicación con el grado de paralelismo obtenido.

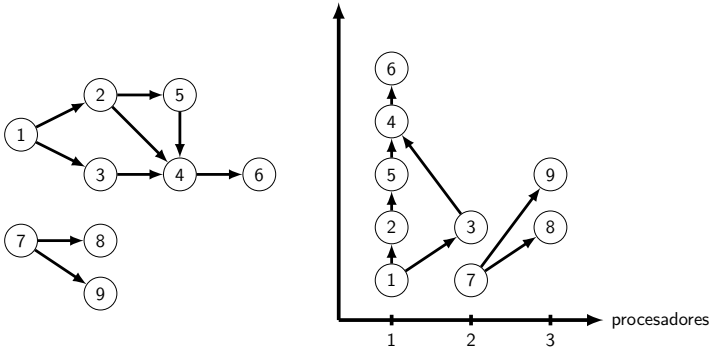
Agenda

Scheduling o Mapeo

- Es el proceso de asignar recursos a múltiples procesos que se ejecutarán en paralelo.
- Determina dónde y cuándo se llevará a cabo cada tarea paralela.
- Las dependencias entre estas tareas juegan un papel crucial en la toma de decisiones sobre la asignación de recursos.

Ejemplo de Scheduling

El scheduling vincula el algoritmo, representado por un grafo, con el hardware, es decir, los procesadores y el tiempo relativo a cada tarea.



Objetivo del Mapeo

El objetivo principal del mapeo es optimizar ciertos criterios clave para mejorar el rendimiento de la aplicación paralela, tales como:

- Tiempo total de ejecución
- Utilización eficiente de los recursos disponibles
- Balance de cargas entre los procesadores

Para lograr esto, se aplican técnicas de programación y algoritmos de planificación que consideran las dependencias de datos y comunicaciones entre tareas, así como la topología de la red en sistemas distribuidos.

Estrategias Comunes de Mapeo

Algunas estrategias comunes en el mapeo incluyen:

- **Asignación en Procesadores Diferentes:** Asignar tareas que pueden ejecutarse simultáneamente a procesadores distintos para maximizar el paralelismo.
- **Agrupación de Tareas Comunicativas:** Asignar tareas que se comunican frecuentemente al mismo procesador o a procesadores cercanos para minimizar la latencia de comunicación.

Agenda

Balance de Cargas

- Factor crítico en el rendimiento de aplicaciones distribuidas que se ejecutan en una red.
- Busca evitar que el desempeño global del sistema se vea comprometido debido a demoras en tareas individuales.
- Adquiere una importancia significativa en entornos no dedicados, donde múltiples tareas compiten por recursos compartidos.

Técnicas de Balance de Cargas

Las técnicas de balance de cargas, también conocidas como “técnicas de despacho”, buscan lograr un uso eficiente de los recursos en aplicaciones distribuidas. Se pueden clasificar en tres categorías principales:

- **Técnicas Estáticas (Planificación):** Las decisiones de despacho se toman tempranamente y se mantienen sin cambios durante la ejecución. Efectivas en entornos poco cargados, pero pueden fallar en ambientes con carga variable.
- **Técnicas Dinámicas (Al Momento del Despacho):** Determinan qué procesador se asigna a una tarea durante la ejecución. Comunes en el modelo maestro-esclavo y efectivas en entornos compartidos con carga variable.
- **Técnicas Adaptativas:** Consideran el estado actual de la red y pueden incorporar herramientas de predicción del futuro. Utilizan técnicas de migración de procesos para aprovechar fluctuaciones de carga.

Parámetros Relevantes para la Determinación de la Carga

La determinación precisa de la carga en una aplicación distribuida implica considerar varios parámetros importantes:

- **Consumo de CPU:** Porcentaje de uso de la CPU o cantidad de operaciones por segundo.
- **Uso de Disco:** Bloques transferidos desde el controlador al dispositivo de disco, indicando la carga de trabajo en el almacenamiento.
- **Tráfico de Red:** Número de paquetes transmitidos y recibidos en la red, proporcionando información sobre la carga en las comunicaciones.

Agenda

Multi-Threading

- Los *threads* o hilos de ejecución son secuencias de instrucciones de cierto proceso.
- Cuando un programa tiene varos hilos, estas secuencias de instrucciones se ejecutan concurrentemente (no necesariamente en paralelo).
- Utilizar varios hilos puede lograr una mayor utilización de los recursos del procesador y una ejecución más eficiente entre distintas tareas.

Ventajas de los Threads

Los threads ofrecen una serie de beneficios significativos:

1. **Mejor Aprovechamiento de los Recursos del Procesador:** Distribución más efectiva de la carga de trabajo.
2. **Rapidez en la Comunicación:** Comunicación más rápida entre threads que entre procesos independientes.
3. **División de Aplicaciones en Módulos Funcionales:** Simplifica el diseño y la programación.
4. **Mejor Rendimiento en Tiempo de Ejecución:** Generalmente un mejor rendimiento comparado con procesos independientes.

Desventajas de los Threads

Existen desventajas en la implementación de threads:

1. **Programación Más Difícil:** Necesidad de gestionar la concurrencia y sincronización.
2. **Dificultad en la Detección de Errores:** Más difícil detectar y depurar errores en entornos con múltiples threads.

Compartición de Recursos entre Threads

- **Compartición del Espacio de Direccionamiento:** Todos los threads dentro de un proceso comparten el mismo espacio de direccionamiento, lo que puede llevar a condiciones de carrera.
- **Ausencia de Protección entre Threads:** No hay protección entre threads; es responsabilidad del programador implementar mecanismos de sincronización.
- **Información Privada de Cada Thread:** Cada thread mantiene su propia información privada, como el contador de programa y la pila.
- **Recursos Compartidos:** Los threads comparten ciertos recursos, como memoria y archivos abiertos, facilitando la comunicación y gestión.

Threads a Nivel de Núcleo del Sistema

Los threads a nivel de núcleo son soportados directamente por el sistema operativo:

- **Ventajas:** El sistema operativo gestiona la creación, planificación y administración de threads, permitiendo un mayor nivel de paralelismo en sistemas multiprocesador.
- **Desventajas:** Cambio de contexto más costoso y necesidad de mantener más estructuras en memoria por parte del sistema operativo.

Threads a Nivel de Usuario

Los threads a nivel de usuario son gestionados mediante una biblioteca de usuario:

- **Ventajas:** Rápido cambio de contexto y no intervención del sistema operativo.
- **Desventajas:** Solo se ejecuta un thread a la vez en entornos multiprocesador, y un bloqueo de un thread puede bloquear a todos los demás.

Agenda

¿Qué es OpenMP?

- OpenMP es una extensión para los lenguajes C y Fortran que facilita la paralelización y el manejo de hilos.
- Permite convertir código secuencial a paralelo de manera incremental.
- Usa directivas del compilador para paralelizar bucles, como las iteraciones de un *for*.
- Facilita la programación paralela en comparación con MPI, ya que no requiere una conversión completa a memoria distribuida.

Compiladores y directivas

- Soportado por compiladores como `gcc` e Intel.
- En C: se usan directivas `#pragma`, como en:

```
#pragma omp parallel for
for (int i=0; i<n; i++) {
    c[i] = a[i] + b[i];
}
```

- Debe enlazarse la biblioteca OpenMP, usando por ejemplo `-fopenmp` en `gcc`.

Distribución de Iteraciones y Schedule

- OpenMP asigna iteraciones de un bucle a distintos hilos automáticamente.
- La directiva `schedule` permite personalizar la asignación de iteraciones.
- Ejemplos de uso:
  - `schedule(static,n)`: asigna bloques de tamaño n a cada hilo.
  - `schedule(dynamic,4)`: asigna bloques de 4 iteraciones a hilos según estén disponibles.

Variables privadas y reducción

- `private(var)`: Define variables privadas para cada hilo.
- Ejemplo:

```
#pragma omp parallel for private(x)
for (int i=0; i<n; i++) {
    x = a[i] + b[i];
    c[i] = x;
}
```

- `reduction(op:var)`: Une resultados parciales de cada hilo en una operación global.
- Ejemplo de suma paralela:

```
int suma = 0;
#pragma omp parallel for reduction(+:suma)
for (int i=0; i<n; i++)
    suma += a[i];
```

Agenda

¿Qué es MPI?

- MPI (Message Passing Interface) es una biblioteca estándar para comunicación entre procesos mediante el paso de mensajes.
- Fue diseñada para sistemas de memoria distribuida, como clusters de computadoras.
- Facilita el paralelismo explícito bajo el modelo SPMD (Single Program, Multiple Data).

Características de MPI

- MPI está diseñado para plataformas de memoria distribuida.
- El programador debe controlar explícitamente el paralelismo.
- El número de procesos es fijo durante la ejecución.
- No permite la creación dinámica de procesos.

Ejemplo básico de MPI

```
ierr = MPI_Init(&argc, &argv); // Inicializa MPI
ierr = MPI_Comm_size(MPI_COMM_WORLD, &npes); // Nro. de procesos
ierr = MPI_Comm_rank(MPI_COMM_WORLD, &iam); // Rango del proceso
// Enviar
ierr = MPI_Send(buffer, count, datatype, destino, tag, comm);
// Recibir
ierr = MPI_Recv(buffer, count, datatype, source, tag, comm, status);
ierr = MPI_Finalize(); // Finaliza MPI
```

Comunicaciones colectivas en MPI

- MPI soporta operaciones colectivas, como:
  - `MPI_Barrier`: Sincroniza todos los procesos.
  - `MPI_Bcast`: Difunde un mensaje desde un proceso raíz a todos los demás.
  - `MPI_Gather`: Recopila mensajes desde todos los procesos y los envía al proceso raíz.
  - `MPI_Scatter`: Distribuye porciones de datos desde un proceso raíz a todos los procesos.
  - `MPI_Reduce`: Aplica una operación de reducción (por ejemplo, suma) a datos distribuidos.

Ventajas de MPI

- Portabilidad: Ejecutable en distintas plataformas sin modificaciones.
- Eficiencia: Utiliza recursos de hardware de forma óptima.
- Flexibilidad: Ofrece un conjunto completo de funciones de comunicación.
- Soporte para Fortran y C.