Arquitecturas de sistemas paralelos

Las arquitecturas de sistemas paralelos son fundamentales para mejorar el rendimiento de las aplicaciones mediante la ejecución simultánea de múltiples operaciones. Estas arquitecturas se dividen en varias categorías, cada una adecuada para diferentes tipos de tareas y aplicaciones. En general, las arquitecturas paralelas se clasifican en función de la organización de la memoria y la interconexión de los procesadores.

En el ámbito de la computación paralela, las arquitecturas de sistemas pueden clasificarse en dos categorías principales basadas en cómo gestionan la memoria: arquitecturas de memoria compartida y arquitecturas de memoria distribuida. Ambas tienen sus ventajas y desventajas, y son adecuadas para diferentes tipos de aplicaciones y entornos de ejecución.

Arquitectura de memoria compartida

En una arquitectura de memoria compartida, todos los procesadores acceden a un único espacio de memoria global. Esta configuración es intuitiva y facilita la programación, ya que los procesos pueden comunicarse y sincronizarse a través de variables compartidas.

Tipos de memoria compartida

- Memoria compartida uniforme (UMA): En UMA (Uniform Memory Access), todos los procesadores tienen igual tiempo de acceso a la memoria principal. Esto significa que la latencia de acceso a la memoria es la misma para todos los procesadores.
  - Ejemplos: Estaciones de trabajo multiprocesador, servidores de gama media.
  - Ventajas: Simplicidad en la programación debido a la uniformidad en el acceso a la memoria. Facilita la coherencia de la caché y la sincronización entre procesadores.
  - Desventajas: No escala bien a un gran número de procesadores debido a la contención de memoria y los cuellos de botella en el bus de memoria.
- Memoria compartida no uniforme (NUMA): En NUMA (Non-Uniform Memory Access), cada procesador tiene su memoria local, y el tiempo de acceso a la memoria varía dependiendo de si se accede a la memoria local o a la memoria de otro procesador.
  - Ejemplos: Servidores de alto rendimiento, sistemas NUMA basados en arquitecturas como AMD EPYC e Intel Xeon.
  - Ventajas: Mejora la escalabilidad al reducir la contención de memoria y permite un acceso más rápido a la memoria local.
  - Desventajas: Mayor complejidad en la programación, ya que se debe gestionar la latencia variable en el acceso a la memoria.

Ventajas y desventajas de la memoria compartida

Ventajas:

- Facilidad de programación: La memoria compartida es más intuitiva para los programadores, ya que se puede usar una única dirección de memoria para la comunicación entre hilos.
- Rendimiento: En sistemas UMA, el acceso uniforme a la memoria puede simplificar la optimización del rendimiento.

- Coherencia de caché: Los sistemas NUMA y UMA gestionan la coherencia de la caché, lo que puede reducir errores en la programación.

Desventajas:

- Escalabilidad: Los sistemas UMA enfrentan limitaciones en la escalabilidad debido a la contención del bus de memoria.
- Complejidad en NUMA: Aunque NUMA mejora la escalabilidad, introduce complejidad en la programación debido a las diferencias en la latencia de acceso a la memoria.
- Costos: Los sistemas de memoria compartida pueden ser más costosos debido a la necesidad de hardware más sofisticado para gestionar la coherencia de la caché y el acceso a la memoria.

Arquitectura de memoria distribuida

En una arquitectura de memoria distribuida, cada procesador tiene su propia memoria local. Los procesadores no pueden acceder directamente a la memoria de otros procesadores. La comunicación entre procesadores se realiza mediante el paso de mensajes.

Tipos de arquitecturas de memoria distribuida

Multicomputadores: Sistemas compuestos por varios nodos de computación, cada uno con su propia CPU y memoria. Los nodos están interconectados mediante una red de alta velocidad.

- Ejemplos: Supercomputadoras, clústeres de computadoras.
- Ventajas: Alta escalabilidad, ya que no hay contención de memoria compartida. Mejor tolerancia a fallos, ya que cada nodo es independiente.
- Desventajas: Programación más compleja debido a la necesidad de gestionar la comunicación y sincronización entre nodos mediante el paso de mensajes.

Clústeres de computadoras: Un tipo específico de multicomputador donde varios sistemas completos (nodos) están conectados a través de una red para trabajar juntos como un solo sistema.

- Ejemplos: Clústeres de alto rendimiento (HPC), clústeres de alta disponibilidad (HA).
- Ventajas: Coste-efectividad al utilizar hardware convencional. Capacidad de combinar recursos de diferentes sistemas para tareas específicas.
- Desventajas: Dependencia de la red para la comunicación, lo que puede introducir latencias significativas.

Ventajas y desventajas de la memoria distribuida

Ventajas:

- Escalabilidad: Permite construir sistemas con un gran número de nodos sin problemas significativos de contención.
- Tolerancia a fallos: La falla de un nodo no afecta necesariamente a los demás, mejorando la fiabilidad del sistema.
- Flexibilidad: Los nodos pueden ser heterogéneos, permitiendo la integración de diferentes tipos de hardware según sea necesario.

Desventajas:

- Complejidad en la programación: Requiere un manejo explícito de la comunicación y sincronización entre nodos, lo que aumenta la complejidad del desarrollo de software.
- Latencia de comunicación: La comunicación entre nodos puede ser lenta debido a la latencia de la red, afectando el rendimiento de las aplicaciones que requieren alta interactividad.

La elección entre una arquitectura de memoria compartida y una de memoria distribuida depende de las características de la aplicación y de los requisitos específicos del sistema.

Aplicaciones adecuadas para memoria compartida:

- Aplicaciones con alta interactividad entre procesos.
- Sistemas donde la facilidad de programación es una prioridad.
- Entornos donde la coherencia de la caché y la sincronización son críticas.

Aplicaciones adecuadas para memoria distribuida:

- Aplicaciones que requieren alta escalabilidad, como simulaciones científicas y análisis de big data.
- Sistemas distribuidos y clústeres donde la tolerancia a fallos es crucial.
- Aplicaciones que pueden tolerar latencias de comunicación más altas y donde la programación paralela se puede manejar mediante el paso de mensajes.

Arquitecturas paralelas híbridas Las arquitecturas paralelas híbridas combinan elementos de arquitecturas de memoria compartida y memoria distribuida para aprovechar las ventajas de ambas y mitigar sus desventajas. Estas arquitecturas se utilizan en sistemas que requieren alta escalabilidad y rendimiento, y son comunes en aplicaciones científicas, de ingeniería y de análisis de datos masivos.

Las arquitecturas híbridas están diseñadas para mejorar la eficiencia y el rendimiento combinando características de sistemas de memoria compartida (como UMA y NUMA) y sistemas de memoria distribuida (como multicomputadores y clústeres). En una arquitectura híbrida, los nodos individuales pueden ser sistemas de memoria compartida que están interconectados mediante una red de alta velocidad para formar un sistema de memoria distribuida.

Un ejemplo típico de una arquitectura híbrida es un clúster de nodos NUMA. Cada nodo es un sistema NUMA, donde múltiples procesadores comparten una memoria local. Los nodos están conectados a través de una red, formando un sistema de memoria distribuida. Este enfoque permite que cada nodo NUMA opere de manera eficiente con baja latencia de memoria local, mientras que la red de interconexión permite la comunicación entre nodos para tareas de gran escala.

Ventajas de las arquitecturas híbridas

- Escalabilidad mejorada: La combinación de memoria compartida y distribuida permite escalar el sistema a un gran número de procesadores sin los problemas de contención de memoria asociados con UMA. La memoria distribuida entre nodos reduce la carga en el bus de memoria y mejora la escalabilidad.
- Flexibilidad: Las arquitecturas híbridas pueden adaptarse a una amplia variedad de aplicaciones, desde tareas intensivas en cálculo hasta aplicaciones que requieren gran

ancho de banda de memoria. Los desarrolladores pueden optimizar las aplicaciones utilizando la memoria local para datos de acceso frecuente y la memoria distribuida para datos que no requieren acceso inmediato.

- Rendimiento: Los nodos NUMA dentro de una arquitectura híbrida permiten un acceso rápido a la memoria local, mejorando el rendimiento de las aplicaciones. La interconexión de alta velocidad entre nodos permite una comunicación eficiente para tareas que requieren intercambio de datos.
- Tolerancia a fallos: La naturaleza distribuida de la memoria permite que el sistema continúe operando incluso si uno o varios nodos fallan. Las arquitecturas híbridas pueden incluir mecanismos de redundancia y recuperación para mejorar la fiabilidad del sistema.

Desventajas de las arquitecturas híbridas

- Complejidad de programación: Los desarrolladores deben gestionar tanto la memoria compartida como la distribuida, lo que aumenta la complejidad del desarrollo de software. La programación paralela en arquitecturas híbridas requiere un profundo conocimiento de los modelos de memoria y las técnicas de sincronización.
- Latencia de comunicación: Aunque la red de interconexión es rápida, la comunicación entre nodos aún puede introducir latencia, afectando el rendimiento de aplicaciones altamente interactivas. Las estrategias de minimización de latencia deben ser cuidadosamente diseñadas y optimizadas.
- Costos: La implementación de arquitecturas híbridas puede ser costosa debido a la necesidad de hardware sofisticado y redes de alta velocidad. Los costos operativos y de mantenimiento también pueden ser altos debido a la complejidad del sistema.

Modelos de programación en arquitecturas híbridas

Para aprovechar las arquitecturas híbridas, se utilizan diversos modelos de programación que combinan técnicas de memoria compartida y distribuida:

- Modelo MPI+OpenMP: Este modelo combina el paso de mensajes (MPI) para la comunicación entre nodos y la memoria compartida (OpenMP) para la paralelización dentro de cada nodo.
  - Ejemplo: En una simulación de dinámica de fluidos, MPI se puede usar para distribuir diferentes regiones del dominio de simulación entre nodos, mientras que OpenMP se usa para paralelizar los cálculos dentro de cada región.
- Modelo UPC (Unified Parallel C): UPC es un modelo de programación paralela que extiende C para incluir tanto memoria compartida como distribuida.
  - Permite a los desarrolladores escribir programas que pueden aprovechar las arquitecturas híbridas sin tener que gestionar explícitamente la comunicación entre nodos.
- Modelo GASNet (Global Address Space Networking): GASNet es una API de comunicaciones diseñada para soportar modelos de programación con espacio de direcciones global.
  - Proporciona una abstracción de comunicación eficiente que permite a los desarrolladores escribir aplicaciones paralelas que pueden escalar en arquitecturas híbridas.

Aplicaciones de las arquitecturas híbridas

Las arquitecturas híbridas son particularmente adecuadas para una amplia gama de aplicaciones científicas y de ingeniería:

Simulaciones científicas:

- Aplicaciones como simulaciones de dinámica molecular, modelado climático y simulaciones de física de alta energía se benefician de la combinación de memoria compartida y distribuida para manejar grandes volúmenes de datos y cálculos intensivos.

Análisis de datos masivos:

- En el análisis de big data, las arquitecturas híbridas permiten procesar grandes conjuntos de datos de manera eficiente mediante la distribución de tareas y datos entre nodos y el uso de memoria compartida para cálculos intensivos.

Inteligencia artificial y aprendizaje automático:

- Los entrenamientos de modelos de aprendizaje profundo pueden aprovechar arquitecturas híbridas para distribuir datos y tareas de entrenamiento entre múltiples nodos, mientras que los cálculos intensivos se realizan dentro de los nodos utilizando memoria compartida.

Ingeniería y diseño asistido por computadora (CAE/CAD):

Las aplicaciones de CAE y CAD, que requieren simulaciones precisas y detalladas, pueden utilizar arquitecturas híbridas para manejar las demandas computacionales y de memoria.

Ejemplos

Un ejemplo real de arquitectura híbrida es un clúster de alto rendimiento (HPC) que utiliza nodos NUMA interconectados mediante una red InfiniBand de alta velocidad. Cada nodo NUMA tiene múltiples procesadores con memoria local compartida, y los nodos están conectados para formar una memoria distribuida global. Este tipo de clúster se utiliza en centros de supercomputación para tareas como la simulación de fenómenos físicos complejos y el análisis de grandes conjuntos de datos.

Características del clúster HPC híbrido:

- Interconexión de alta velocidad:
  - La red InfiniBand proporciona una latencia baja y un alto ancho de banda para la comunicación entre nodos, mejorando el rendimiento global del sistema.
- Gestión de memoria eficiente:
  - Los desarrolladores pueden aprovechar la memoria local de los nodos NUMA para datos de acceso frecuente y utilizar la memoria distribuida para datos menos críticos.

Soporte para diversos modelos de programación:

- El clúster puede soportar modelos de programación como MPI+OpenMP, UPC y GASNet, proporcionando flexibilidad para los desarrolladores. Escalabilidad y Rendimiento:

La arquitectura híbrida permite escalar el clúster a miles de nodos, proporcionando un rendimiento sustancial para aplicaciones de gran escala.

Técnicas orientadas a computación paralela y distribuida

Paralelismo de datos

El paralelismo de datos implica dividir grandes conjuntos de datos en partes más pequeñas y procesarlas simultáneamente en diferentes procesadores. Esta técnica es efectiva en aplicaciones como el procesamiento de imágenes, simulaciones científicas, y análisis de grandes volúmenes de datos.

Paralelismo de tareas

El paralelismo de tareas divide un programa en tareas o hilos que se ejecutan en paralelo. Cada tarea puede realizar una operación diferente, permitiendo la ejecución concurrente de múltiples operaciones. Esta técnica es útil en aplicaciones como servidores web y procesamiento en tiempo real.

Modelos de programación paralela

- Modelo de paso de mensajes (MPI): Utilizado en sistemas de memoria distribuida, donde los procesadores se comunican enviando y recibiendo mensajes. MPI es un estándar ampliamente utilizado en aplicaciones de HPC.
- Modelo de memoria compartida (OpenMP): Utilizado en sistemas de memoria compartida, permite la creación de aplicaciones paralelas mediante directivas en el código fuente. OpenMP simplifica la programación paralela en sistemas SMP y multinúcleo.
- Modelo híbrido: Combina MPI y OpenMP para aprovechar las ventajas de ambos modelos en sistemas NUMA o clústeres heterogéneos.

Ejercicios

- Explica la diferencia entre UMA (Uniform Memory Access) y NUMA (Non-Uniform Memory Access).
- Describe las ventajas y desventajas de la memoria compartida en comparación con la memoria distribuida.
- ¿Qué es la coherencia de caché y por qué es importante en sistemas de memoria compartida?
- Describe el proceso de comunicación entre nodos en un sistema de memoria distribuida.
- Explica las técnicas comunes para optimizar la comunicación en sistemas de memoria distribuida.
- Discute los desafíos de programación en sistemas de memoria distribuida.
- Define qué es una arquitectura de memoria híbrida y da ejemplos de sistemas que la utilizan.
- Explica cómo se puede combinar MPI y OpenMP en una arquitectura híbrida para mejorar el rendimiento.

- Discute las consideraciones de diseño al desarrollar aplicaciones para arquitecturas híbridas.

```
## Tus respuestas
```

1 . Escribe un programa en Python que use la librería multiprocessing para simular múltiples procesos accediendo y modificando una variable compartida. Usa un Manager para manejar el acceso seguro a la variable compartida.

```
from multiprocessing import Process, Manager

def increment(shared_dict, key):
    for _ in range(1000):
        shared_dict[key] += 1

if __name__ == "__main__":
    manager = Manager()
    shared_dict = manager.dict()
    shared_dict["counter"] = 0

    processes = []
    for _ in range(4):
        p = Process(target=increment, args=(shared_dict, "counter"))
        processes.append(p)
        p.start()

    for p in processes:
        p.join()

    print(f"Final counter value: {shared_dict['counter']}")
```

2 . Usa la librería mpi4py para simular la comunicación entre nodos en un sistema de memoria distribuida. Crea un programa que envíe y reciba mensajes entre múltiples procesos.

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    data = "Hello from rank 0"
    for i in range(1, size):
        comm.send(data, dest=i)
else:
    data = comm.recv(source=0)
    print(f"Rank {rank} received data: {data}")
```

3 . Combina MPI y OpenMP en un programa Python utilizando mpi4py y threading. Simula una tarea que se distribuye entre nodos y luego se paraleliza dentro de cada nodo.

```python
from mpi4py import MPI
from threading import Thread

def thread_task(rank):
    print(f"Thread in rank {rank} is running")

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

threads = []
for _ in range(4):
    t = Thread(target=thread_task, args=(rank,))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

if rank == 0:
    print("All threads completed.")
```

4 . Escribe un programa en C que use pthread para crear múltiples hilos que accedan y modifiquen una variable global compartida de manera segura usando mutex.

```c
#include <pthread.h>
#include <stdio.h>

int counter = 0;
pthread_mutex_t lock;

void* increment(void* arg) {
    for (int i = 0; i < 1000; i++) {
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main() {
    pthread_t threads[4];
    pthread_mutex_init(&lock, NULL);

    for (int i = 0; i < 4; i++) {
        pthread_create(&threads[i], NULL, increment, NULL);
    }

    for (int i = 0; i < 4; i++) {
        pthread_join(threads[i], NULL);
```

```
    }

    pthread_mutex_destroy(&lock);
    printf("Final counter value: %d\n", counter);
    return 0;
}
```

5 . Usa MPI en C para crear un programa que envíe y reciba mensajes entre múltiples procesos.

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        char message[] = "Hello from rank 0";
        for (int i = 1; i < size; i++) {
            MPI_Send(message, sizeof(message), MPI_CHAR, i, 0,
MPI_COMM_WORLD);
        }
    } else {
        char message[20];
        MPI_Recv(message, 20, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        printf("Rank %d received message: %s\n", rank, message);
    }

    MPI_Finalize();
    return 0;
}
```

6 . Escribe un programa en Python que use multiprocessing.Pool para sumar los elementos de una lista en paralelo. Divide la lista en partes iguales y asigna cada parte a un proceso.

```
from multiprocessing import Pool

def sum_segment(segment):
    return sum(segment)

if __name__ == "__main__":
    data = list(range(1000000))  # Lista de un millón de elementos
    num_processes = 4
    segment_size = len(data) // num_processes
```

```
    segments = [data[i * segment_size:(i + 1) * segment_size] for i in
range(num_processes)]

    with Pool(num_processes) as pool:
        results = pool.map(sum_segment, segments)

    total_sum = sum(results)
    print(f"Total sum: {total_sum}")
```

7 . Implementa un patrón productor-consumidor usando multiprocessing.Queue en Python. Crea varios procesos productores que pongan datos en la cola y varios procesos consumidores que lean y procesen esos datos.

```
from multiprocessing import Process, Queue
import time
import random

def producer(queue):
    for _ in range(10):
        item = random.randint(1, 100)
        queue.put(item)
        print(f"Produced {item}")
        time.sleep(random.random())

def consumer(queue):
    while True:
        item = queue.get()
        if item is None:  # Sentinel value to indicate end of
processing
            break
        print(f"Consumed {item}")
        time.sleep(random.random())

if __name__ == "__main__":
    queue = Queue()

    producers = [Process(target=producer, args=(queue,)) for _ in
range(2)]
    consumers = [Process(target=consumer, args=(queue,)) for _ in
range(2)]

    for p in producers:
        p.start()

    for c in consumers:
        c.start()

    for p in producers:
        p.join()
```

```
    # Add sentinel values to the queue to signal consumers to exit
    for _ in consumers:
        queue.put(None)

    for c in consumers:
        c.join()
```

8 . Usa mpi4py para implementar un broadcast donde un proceso envía datos a todos los demás procesos en el comunicador.

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'key1': 'value1', 'key2': 'value2'}
else:
    data = None

data = comm.bcast(data, root=0)

print(f"Rank {rank} received data: {data}")
```

9 . Implementa una reducción (reduce) con mpi4py para sumar números distribuidos entre varios procesos.

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Cada proceso tiene un número diferente
number = rank + 1

total_sum = comm.reduce(number, op=MPI.SUM, root=0)

if rank == 0:
    print(f"Total sum: {total_sum}")
```

10 . Combina MPI y threading para distribuir una lista de datos entre nodos y luego procesarla en paralelo dentro de cada nodo.

```
from mpi4py import MPI
from threading import Thread

def process_data(data_segment):
    result = [x ** 2 for x in data_segment]
    print(f"Processed segment: {result}")
```

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

data = None
if rank == 0:
    data = list(range(100))
    segment_size = len(data) // size
    data_segments = [data[i * segment_size:(i + 1) * segment_size] for
i in range(size)]
else:
    data_segments = None

data_segment = comm.scatter(data_segments, root=0)

threads = []
for i in range(4):
    t = Thread(target=process_data, args=(data_segment[i::4],))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

comm.Barrier()

if rank == 0:
    print("All data processed.")
```

11 . Usa MPI para recolectar y combinar los resultados de datos procesados en paralelo utilizando OpenMP en cada nodo.

```
from mpi4py import MPI
from threading import Thread

def compute_square(value, result_list, index):
    result_list[index] = value ** 2

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

data = None
if rank == 0:
    data = list(range(1, 101))
    segment_size = len(data) // size
    data_segments = [data[i * segment_size:(i + 1) * segment_size] for
i in range(size)]
else:
```

```
    data_segments = None

data_segment = comm.scatter(data_segments, root=0)

results = [0] * len(data_segment)
threads = []
for i in range(len(data_segment)):
    t = Thread(target=compute_square, args=(data_segment[i], results,
i))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

gathered_results = comm.gather(results, root=0)

if rank == 0:
    final_results = [item for sublist in gathered_results for item in
sublist]
    print(f"Final results: {final_results}")

### Tus respuestas
```

Multiprocesadores

Los multiprocesadores son sistemas que utilizan múltiples CPUs en un solo sistema de memoria compartida. Estas CPUs pueden estar en un solo chip (multinúcleo) o en varios chips (multiprocesador). Los multiprocesadores pueden mejorar significativamente el rendimiento de las aplicaciones mediante el paralelismo a nivel de tarea o de hilo.

Tipos de multiprocesadores

- Sistemas simétricos de multiprocesamiento (SMP): En SMP, todos los procesadores son iguales y comparten la memoria principal de manera equitativa. Cada procesador tiene acceso directo a toda la memoria y a los dispositivos de E/S. Los sistemas SMP son ideales para aplicaciones que requieren alta interactividad entre procesos.
- Sistemas asimétricos de multiprocesamiento (AMP): En AMP, los procesadores no son iguales. Un procesador principal controla el sistema y asigna tareas a los procesadores secundarios. Este modelo es menos común y se utiliza en sistemas embebidos o en aplicaciones específicas.

Ventajas de los Multiprocesadores

- Mejoran el rendimiento general al ejecutar múltiples tareas simultáneamente.
- Simplifican la programación comparado con sistemas de memoria distribuida.
- Buena escalabilidad en sistemas NUMA.

Multicomputadores

Los multicomputadores, también conocidos como sistemas de memoria distribuida, consisten en múltiples nodos de computación, cada uno con su propia CPU y memoria, interconectados por una red de alta velocidad. Estos sistemas son altamente escalables y se utilizan en aplicaciones que requieren un gran poder de cómputo, como simulaciones científicas y análisis de big data.

Ventajas de los multicomputadores

- Alta escalabilidad debido a la naturaleza independiente de los nodos.
- Mayor tolerancia a fallos, ya que la falla de un nodo no afecta a los demás.

Desventajas de los multicomputadores

- Programación compleja debido a la necesidad de gestionar la comunicación entre nodos.
- Latencias de comunicación más altas comparadas con sistemas de memoria compartida.

Clústeres

Los clústeres son un tipo de arquitectura de multicomputador donde varios sistemas completos (nodos) están conectados a través de una red para trabajar juntos como un solo sistema. Los clústeres pueden ser homogéneos (todos los nodos son iguales) o heterogéneos (nodos con diferentes capacidades).

Tipos de clústeres

- Clústeres de alto rendimiento (HPC): Diseñados para maximizar el rendimiento y son utilizados en simulaciones científicas, análisis de datos, y otras aplicaciones que requieren gran poder de cómputo.
- Clústeres de alta disponibilidad (HA): Enfocados en proporcionar alta disponibilidad y tolerancia a fallos. Se utilizan en aplicaciones críticas donde el tiempo de inactividad debe minimizarse.
- Clústeres de balanceo de carga: Distribuyen la carga de trabajo entre varios nodos para mejorar el rendimiento y la disponibilidad de servicios web y aplicaciones empresariales.

Ventajas de los clústeres

- Alta escalabilidad y flexibilidad.
- Costo-efectividad al utilizar hardware convencional.
- Capacidad de combinar recursos de diferentes sistemas para tareas específicas. Desventajas de los Clústeres Complejidad en la configuración y gestión. Dependencia de la red para la comunicación entre nodos, lo que puede introducir latencias.

Ejercicios

- Explica las diferencias fundamentales entre un multiprocesador y un multicomputador en términos de arquitectura, comunicación entre procesadores y aplicaciones típicas.

- Discute los diferentes modelos de consistencia de memoria en sistemas de multiprocesadores. Compara y contrasta la consistencia secuencial, la consistencia causal y la consistencia eventual, proporcionando ejemplos de situaciones en las que cada uno podría ser adecuado.
- Define el problema de coherencia de caché en sistemas multiprocesadores y describe al menos dos protocolos de coherencia de caché (como MESI y MOESI). Explica cómo estos protocolos ayudan a mantener la coherencia y los posibles problemas de rendimiento asociados. Paralelismo a Nivel de Instrucción (ILP)
- Describe el concepto de paralelismo a nivel de instrucción (ILP) y cómo los multiprocesadores pueden explotarlo. Discute técnicas como la ejecución fuera de orden y la predicción de saltos, y cómo estas técnicas afectan el rendimiento.
- Describe las diferentes topologías de red utilizadas en multicomputadores (como malla, hipercubo, anillo y estrella). Analiza las ventajas y desventajas de cada topología en términos de latencia, ancho de banda y tolerancia a fallos.
- Compara y contrasta los protocolos de comunicación punto a punto y basados en mensajes en multicomputadores. Proporciona ejemplos de situaciones en las que uno podría ser más adecuado que el otro.
- Define el concepto de escalabilidad en el contexto de multicomputadores y discute los factores que afectan la escalabilidad, como la latencia de comunicación, el ancho de banda y el balance de carga. Proporciona ejemplos de cómo diseñar un sistema multicomputador escalable.
- Explica los desafíos de la sincronización en sistemas distribuidos y describe al menos dos algoritmos de sincronización (como el algoritmo de Lamport y el algoritmo de Ricart-Agrawala). Discute cómo estos algoritmos aseguran la exclusión mutua y las posibles desventajas.
- Describe los componentes clave en el diseño de un clúster de alto rendimiento, incluyendo nodos de computación, redes de interconexión, almacenamiento y software de gestión. Discute las decisiones de diseño que afectan el rendimiento y la escalabilidad del clúster. Modelos de Programación para Clústeres
- Compara y contrasta los modelos de programación para clústeres, como MPI (Message Passing Interface) y OpenMP (Open Multi-Processing). Discute las ventajas y desventajas de cada modelo y proporciona ejemplos de aplicaciones típicas.
- Explica la importancia de la tolerancia a fallos en clústeres y describe las técnicas comunes para lograrla, como la replicación de datos, el checkpointing y la migración de tareas. Discute los desafíos asociados con cada técnica.
- Describe los métodos utilizados para evaluar el rendimiento de un clúster, como benchmarks (por ejemplo, LINPACK, HPCG) y métricas (por ejemplo, FLOPS, latencia, ancho de banda). Explica cómo interpretar los resultados y utilizarlos para optimizar el rendimiento del clúster.
- Análisis de escalabilidad_ Un sistema multiprocesador tiene 8 núcleos y se observa que una aplicación que tarda 100 segundos en ejecutarse en un solo núcleo tarda 20 segundos en ejecutarse en 8 núcleos. Calcula la aceleración (speedup) y la eficiencia del sistema. ¿Es el sistema escalable? Justifica tu respuesta.
- Describe el problema de la cena de los filósofos y cómo se aplica a los sistemas de multiprocesadores. Propón una solución utilizando semáforos o monitores y discute las posibles situaciones de deadlock y cómo evitarlas.

- Investiga un caso de estudio real de un clúster de alto rendimiento utilizado en una aplicación científica o industrial. Describe la arquitectura del clúster, el tipo de aplicaciones que ejecuta, y los desafíos y soluciones implementadas en términos de rendimiento y escalabilidad.
- Realiza una evaluación comparativa teórica de dos arquitecturas de clústeres diferentes (por ejemplo, uno basado en CPU y otro basado en GPU). Discute las ventajas y desventajas de cada arquitectura para diferentes tipos de aplicaciones.

```
## Tus respuestas
```

1 . Implementa la multiplicación de dos matrices grandes en paralelo utilizando la librería multiprocessing.

```
from multiprocessing import Pool
import numpy as np

def matrix_multiply_segment(args):
    A_segment, B = args
    return np.dot(A_segment, B)

if __name__ == "__main__":
    A = np.random.rand(1000, 1000)
    B = np.random.rand(1000, 1000)

    num_processes = 4
    segment_size = A.shape[0] // num_processes
    segments = [(A[i * segment_size:(i + 1) * segment_size], B) for i
in range(num_processes)]

    with Pool(num_processes) as pool:
        results = pool.map(matrix_multiply_segment, segments)

    C = np.vstack(results)
    print("Matrix multiplication completed.")
```

2 . Implementa la suma de dos vectores grandes distribuidos entre varios nodos utilizando mpi4py.

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

vector_size = 1000000
local_size = vector_size // size

if rank == 0:
    A = np.random.rand(vector_size)
```

```
    B = np.random.rand(vector_size)
else:
    A = None
    B = None

local_A = np.empty(local_size, dtype='d')
local_B = np.empty(local_size, dtype='d')

comm.Scatter(A, local_A, root=0)
comm.Scatter(B, local_B, root=0)

local_C = local_A + local_B

if rank == 0:
    C = np.empty(vector_size, dtype='d')
else:
    C = None

comm.Gather(local_C, C, root=0)

if rank == 0:
    print("Vector addition completed.")
```

3 . Implementa una simulación de Monte Carlo para calcular el valor de Pi en paralelo en un clúster utilizando mpi4py.

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

num_samples = 1000000
local_samples = num_samples // size

np.random.seed(rank)
local_count = 0

for _ in range(local_samples):
    x, y = np.random.rand(2)
    if x**2 + y**2 <= 1.0:
        local_count += 1

total_count = comm.reduce(local_count, op=MPI.SUM, root=0)

if rank == 0:
    pi_estimate = (4.0 * total_count) / num_samples
    print(f"Estimated Pi: {pi_estimate}")
```

4 . Divide y procesa un gran archivo de texto en un clúster, donde cada nodo procesa una parte del archivo y luego los resultados se combinan.

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    with open('large_text_file.txt', 'r') as file:
        lines = file.readlines()
else:
    lines = None

lines = comm.scatter(lines, root=0)

local_word_count = sum(len(line.split()) for line in lines)

total_word_count = comm.reduce(local_word_count, op=MPI.SUM, root=0)

if rank == 0:
    print(f"Total word count: {total_word_count}")
```

5 . Usa pthread para dividir un arreglo en segmentos y sumarlos en paralelo

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 4
#define ARRAY_SIZE 1000000

int array[ARRAY_SIZE];
long long sum = 0;
pthread_mutex_t mutex;

void* sum_segment(void* arg) {
    int start = (int)arg * (ARRAY_SIZE / NUM_THREADS);
    int end = start + (ARRAY_SIZE / NUM_THREADS);
    long long local_sum = 0;

    for (int i = start; i < end; i++) {
        local_sum += array[i];
    }

    pthread_mutex_lock(&mutex);
    sum += local_sum;
    pthread_mutex_unlock(&mutex);
```

```
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    pthread_mutex_init(&mutex, NULL);

    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = rand() % 100;
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, sum_segment, (void*)i);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    printf("Total sum: %lld\n", sum);
    return 0;
}
```

6 . Implementa la suma de dos vectores grandes distribuidos entre varios nodos utilizando MPI en C.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int vector_size = 1000000;
    int local_size = vector_size / size;

    double* local_A = (double*)malloc(local_size * sizeof(double));
    double* local_B = (double*)malloc(local_size * sizeof(double));
    double* local_C = (double*)malloc(local_size * sizeof(double));

    for (int i = 0; i < local_size; i++) {
        local_A[i] = rank + 1;
        local_B[i] = rank + 2;
    }
```

```
    MPI_Allreduce(local_A, local_C, local_size, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);

    for (int i = 0; i < local_size; i++) {
        local_C[i] = local_A[i] + local_B[i];
    }

    free(local_A);
    free(local_B);
    free(local_C);

    MPI_Finalize();
    return 0;
}
```

7 . Implementa una simulación de Monte Carlo para calcular el valor de Pi en paralelo en un clúster utilizando MPI en C.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    long long num_samples = 1000000;
    long long local_samples = num_samples / size;
    long long local_count = 0;
    unsigned int seed = rank;

    for (long long i = 0; i < local_samples; i++) {
        double x = (double)rand_r(&seed) / RAND_MAX;
        double y = (double)rand_r(&seed) / RAND_MAX;
        if (x * x + y * y <= 1.0) {
            local_count++;
        }
    }

    long long total_count;
    MPI_Reduce(&local_count, &total_count, 1, MPI_LONG_LONG_INT,
MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        double pi_estimate = (4.0 * total_count) / num_samples;
        printf("Estimated Pi: %f\n", pi_estimate);
    }
```

```
    MPI_Finalize();
    return 0;
}
```

Tus respuestas