Análisis de desempeño y escalabilidad en computación paralela

El análisis de desempeño y escalabilidad en computación paralela es crucial para entender cómo los algoritmos y sistemas pueden aprovechar múltiples procesadores para mejorar el rendimiento. Este análisis incluye conceptos fundamentales como la Ley de Amdahl y la Ley de Gustafson, así como medidas clave como el `speedup` y la eficiencia. También es esencial considerar el balanceo de carga y las técnicas para medir y optimizar el rendimiento.

Ley de Amdahl

La Ley de Amdahl, formulada por Gene Amdahl en 1967, es una fórmula utilizada para encontrar el límite superior de aceleración de un programa paralelo. Esta ley es fundamental para entender las limitaciones inherentes al paralelismo.

La ley de Amdahl se expresa matemáticamente como:

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}}$$

Donde:

- $S(N)$ es el speedup con N procesadores.
- P es la fracción del programa que puede paralelizarse.
- $1 - P$ es la fracción del programa que debe ejecutarse secuencialmente.

Speedup (S): Es la medida de cuánto más rápido se ejecuta un programa con múltiples procesadores en comparación con uno solo.

Fracción paralelizable (P): Indica la porción del programa que puede beneficiarse del paralelismo.

Límite de aceleración: A medida que N aumenta, el término $\frac{P}{N}$ se vuelve insignificante, y el speedup máximo está limitado por $\frac{1}{1 - P}$.

Por ejemplo, si el 90% de un programa puede paralelizarse $P = 0.9$, el speedup máximo es 10, sin importar cuántos procesadores se utilicen.

Un caso real donde se aplicó la Ley de Amdahl es el desarrollo del software para simulaciones meteorológicas. En este contexto, solo una fracción del cálculo (aproximadamente el 80%) puede paralelizarse debido a las dependencias inherentes en los datos meteorológicos.

En la práctica, esto significa que incluso con un número muy grande de procesadores, el speedup máximo está limitado por la porción secuencial del programa. Por ejemplo, con $P = 0.8$, el speedup máximo es:

$$S(N \to \infty) = \frac{1}{1 - 0.8} = 5$$

Esto implica que, sin importar cuántos procesadores se utilicen, la aceleración no puede superar 5 veces el tiempo de ejecución secuencial.

Ley de Gustafson

La ley de Gustafson, propuesta por John L. Gustafson en 1988, ofrece una perspectiva diferente y más optimista sobre el paralelismo, sugiriendo que la velocidad de un programa paralelo puede aumentar linealmente con el número de procesadores.

La ley de Gustafson se expresa como:

$$S(N) = N - (N - 1) \cdot (1 - P)$$

Donde:

- $S(N)$ es el speedup con N procesadores.
- P es la fracción del programa que puede paralelizarse.

Escalabilidad: Gustafson argumenta que los problemas grandes pueden ser escalados para aprovechar el paralelismo, con un crecimiento en la porción paralelizable.

Aceleración escalable: A diferencia de la ley de Amdahl, esta fórmula sugiere que el speedup puede crecer indefinidamente con N, siempre y cuando P aumente con el tamaño del problema.

Por ejemplo, si $P = 0.9$ y $N = 100$, entonces $S(100) = 100 - 99 \cdot 0.1 = 91$.

Un caso real de aplicación de la Ley de Gustafson se observa en proyectos de renderización gráfica en la industria cinematográfica, donde los problemas pueden ser escalados para aprovechar el paralelismo. En la práctica, si se tiene $P = 0.95$ y $N = 100$:

$$S(100) = 100 - 99 \cdot 0.05 = 95.05$$

Esto muestra que el speedup puede crecer casi linealmente con el número de procesadores, siempre que se pueda escalar la parte paralelizable del problema.

Experimentación: Para comparar ambas leyes en la práctica, se pueden realizar experimentos utilizando diferentes algoritmos y tamaños de problema. Supongamos que ejecutamos un algoritmo de procesamiento de imágenes con diferentes fracciones paralelizables P y medimos el speedup en una máquina con hasta 64 procesadores.

Escenario 1 - Baja paralelización (P = 0.5):

- Ley de Amdahl: El speedup máximo será limitado significativamente:

$$S(64) = \frac{1}{0.5 + \frac{0.5}{64}} = 1.96$$

- Ley de Gustafson: El speedup será:

$$S(64) = 64 - 63 \cdot 0.5 = 32.5$$

Escenario 2 - Alta paralelización (P = 0.9):

- Ley de Amdahl: El speedup máximo será mayor:

$$S(64)=\frac{1}{0.1+\frac{0.9}{64}}=6.4$$

- Ley de Gustafson: El speedup será:

$$S(64)=64-63\cdot 0.1=57.3$$

Estos resultados experimentales demostrarían que la Ley de Amdahl subestima el potencial de escalabilidad en aplicaciones altamente paralelizables, mientras que la Ley de Gustafson proporciona una estimación más optimista.

Speedup y eficiencia

El speedup y la eficiencia son métricas clave para evaluar el desempeño de algoritmos paralelos.

Speedup

El speedup S es la razón del tiempo de ejecución del mejor algoritmo secuencial T_s al tiempo de ejecución del algoritmo paralelo T_p:

$$S=\frac{T_s}{T_p}$$

Esta métrica indica cuántas veces es más rápido el programa paralelo comparado con su contraparte secuencial.

Eficiencia

La eficiencia E es la razón del speedup al número de procesadores N:

$$E=\frac{S}{N}$$

Esta métrica indica qué tan bien se utilizan los procesadores disponibles. Una eficiencia de 1 (o 100%) indica uso óptimo.

Overhead de comunicación

Impacto del overhead:

El overhead de comunicación se refiere al tiempo adicional requerido para coordinar y transferir datos entre procesadores en un sistema paralelo. Este overhead puede ser un factor limitante significativo en el rendimiento del paralelismo. Por ejemplo, en aplicaciones de simulación molecular, donde las partículas deben comunicarse entre sí en cada paso del tiempo, el overhead de comunicación puede consumir una porción considerable del tiempo total de ejecución.

Minimización del overhead:

Para minimizar este overhead, se pueden utilizar técnicas como:

- **Comunicación Asincrónica:** Permitir que los procesadores continúen trabajando mientras esperan datos de otros procesadores.
- **Agrupamiento de Mensajes:** Enviar varios mensajes en uno solo para reducir la frecuencia de comunicación.
- **Topologías de Red Eficientes:** Usar topologías de red que minimicen el número de saltos entre procesadores, como la topología de malla o el hipercubo.

Latencia y ancho de banda

Latencia: La latencia es el tiempo que tarda un mensaje en viajar desde su fuente hasta su destino. En sistemas distribuidos, la latencia puede ser afectada por la distancia física entre nodos, la congestión de la red, y las colas en los routers. En aplicaciones de alta frecuencia, como el comercio de alta frecuencia, una latencia baja es crucial para la toma de decisiones rápidas.

Ancho de banda: El ancho de banda se refiere a la cantidad de datos que pueden ser transferidos por la red en un tiempo dado. En aplicaciones como el procesamiento de grandes volúmenes de datos (Big Data), un alto ancho de banda es esencial para mover datos rápidamente entre nodos de procesamiento.

Optimización: Para optimizar tanto la latencia como el ancho de banda, se pueden emplear estrategias como:

- **Redes de alta velocidad:** Utilizar redes de alta velocidad como Infiniband o Ethernet de 10/40/100 Gbps.
- **Compresión de datos:** Reducir la cantidad de datos a transferir mediante técnicas de compresión.
- **Estrategias de colocación:** Colocar datos y tareas en nodos cercanos para minimizar la distancia de comunicación.

Experimentos y simulaciones

Para ilustrar el impacto del overhead y las métricas de latencia y ancho de banda, se pueden diseñar y ejecutar experimentos en un cluster de computación.

Experimento 1 - Overhead de comunicación:

- **Setup:** Implementar un algoritmo de multiplicación de matrices en un cluster con diferentes topologías de red.
- **Medición:** Comparar el tiempo de ejecución total y el tiempo dedicado a la comunicación en topologías de malla, anillo, y hipercubo.
- **Resultados esperados:** Identificar la topología con menor overhead y analizar cómo la elección de topología afecta el rendimiento del algoritmo.

Experimento 2 - Latencia y ancho de banda:

- **Setup:** Configurar un entorno distribuido para procesamiento de datos con diferentes configuraciones de red (e.g., Ethernet de 1 Gbps vs. 100 Gbps).
- **Medición:** Medir el tiempo de ejecución de tareas de procesamiento de datos bajo diferentes configuraciones de red.

- **Resultados esperados:** Evaluar el impacto de la latencia y el ancho de banda en el tiempo de procesamiento y determinar las configuraciones óptimas para diferentes tipos de tareas.

Estos experimentos proporcionarían datos empíricos valiosos para entender cómo el overhead de comunicación, la latencia, y el ancho de banda influyen en el rendimiento de aplicaciones paralelas y distribuidas.

Speedup y eficiencia en entornos heterogéneos

Introducción

El análisis de speedup y eficiencia en entornos heterogéneos es una tarea compleja que involucra la integración de diversos tipos de procesadores y aceleradores. A medida que las aplicaciones requieren un mayor rendimiento, la utilización de GPUs, FPGAs y otros aceleradores se ha convertido en una práctica común. Estos aceleradores ofrecen ventajas significativas en términos de procesamiento paralelo y rendimiento energético. Sin embargo, la gestión de estos recursos en un sistema heterogéneo presenta desafíos únicos en términos de escalabilidad y eficiencia.

Aceleradores hardware

Unidades de procesamiento gráfico (GPUs)

Las GPUs están diseñadas para manejar operaciones de procesamiento paralelo masivo, lo que las hace ideales para tareas que pueden ser divididas en múltiples sub-tareas ejecutadas en paralelo. Las GPUs son particularmente eficaces en aplicaciones de aprendizaje profundo, simulaciones científicas y renderizado gráfico.

Ejemplo de uso de GPU:

```
import tensorflow as tf

# Configuración de un modelo simple de TensorFlow para ejecutarse en
una GPU
with tf.device('/GPU:0'):
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(512, activation='relu',
input_shape=(784,)),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(10, activation='softmax')
    ])

    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

# Supongamos que `train_images` y `train_labels` son los datos de
entrenamiento
model.fit(train_images, train_labels, epochs=5)
```

El uso de GPUs puede mejorar el speedup significativamente debido a su capacidad para manejar miles de hilos en paralelo, reduciendo así el tiempo de procesamiento en comparación con CPUs tradicionales.

Field-Programmable Gate Arrays (FPGAs)

Las FPGAs ofrecen un enfoque diferente al procesamiento paralelo. Estos dispositivos son reconfigurables y pueden ser programados para realizar tareas específicas con una eficiencia energética muy alta. Las FPGAs son utilizadas en aplicaciones donde se requiere procesamiento en tiempo real y latencia ultra baja, como en telecomunicaciones, procesamiento de señales y sistemas embebidos.

Ejemplo de uso de FPGA:

```
-- Ejemplo de un módulo simple en VHDL para sumar dos números
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Adder is
    Port ( A : in STD_LOGIC_VECTOR (31 downto 0);
           B : in STD_LOGIC_VECTOR (31 downto 0);
           SUM : out STD_LOGIC_VECTOR (31 downto 0));
end Adder;

architecture Behavioral of Adder is
begin
    SUM <= A + B;
end Behavioral;
```

La ventaja de las FPGAs radica en su capacidad para ser optimizadas específicamente para la tarea que se está ejecutando, ofreciendo un balance único entre rendimiento y consumo energético.

Otros aceleradores

Además de GPUs y FPGAs, otros aceleradores como las unidades de procesamiento tensorial (TPUs) y los procesadores de señales digitales (DSPs) también juegan un papel crucial en entornos heterogéneos. Las TPUs, desarrolladas por Google, están optimizadas para el entrenamiento y la inferencia de modelos de aprendizaje profundo. Los DSPs, por otro lado, son utilizados principalmente en aplicaciones de procesamiento de señales, como audio y video.

Speedup en entornos heterogéneos

El speedup es una métrica clave que mide la mejora en el rendimiento cuando se utiliza un sistema paralelo o distribuido en comparación con un sistema secuencial. En entornos heterogéneos, el cálculo del speedup se complica debido a la diversidad de arquitecturas y capacidades de procesamiento.

Modelo de speedup para GPUs

El speedup en sistemas que utilizan GPUs puede ser modelado utilizando la Ley de Amdahl modificada para tener en cuenta la porción de la tarea que puede ser paralelizada y el overhead de comunicación entre CPU y GPU.

$$S_{GPU} = \frac{1}{(1-P) + \frac{P}{S_{GPU}} + O_{comm}}$$

donde:

- P es la fracción de la tarea que puede ser paralelizada.
- S_{GPU} es el speedup de la porción paralelizada en la GPU.
- O_{comm} es el overhead de comunicación entre CPU y GPU.

Modelo de speedup para FPGAs

Para FPGAs, el speedup también depende de la eficiencia del mapeo de la tarea al hardware reconfigurable y del overhead asociado con la reconfiguración del hardware.

$$S_{FPGA} = \frac{1}{(1-P) + \frac{P}{S_{FPGA}} + O_{config}}$$

donde:

- P es la fracción de la tarea que puede ser paralelizada.
- S_{FPGA} es el speedup de la porción paralelizada en la FPGA.
- O_{config} es el overhead de reconfiguración del FPGA.

Eficiencia en entornos heterogéneos

La eficiencia en sistemas heterogéneos es otra métrica importante que mide la utilización de los recursos disponibles. La eficiencia puede ser afectada por diversos factores, incluyendo la carga de trabajo, la distribución de tareas y el overhead de comunicación.

Factores que afectan la eficiencia

1. **Balance de carga:** En entornos heterogéneos, es crucial distribuir las tareas de manera que todos los recursos se utilicen de manera equilibrada. El balance de carga puede ser desafiante debido a las diferencias en las capacidades de procesamiento de los diferentes tipos de aceleradores.

2. **Overhead de comunicación:** La comunicación entre diferentes tipos de procesadores y aceleradores puede introducir overhead significativo, afectando la eficiencia global del sistema. Optimizar la comunicación y minimizar el overhead es esencial para mantener alta eficiencia.

3. **Granularidad de las tareas:** La granularidad de las tareas (el tamaño de las sub-tareas en que se divide una tarea) debe ser adecuada para el tipo de acelerador

utilizado. Las GPUs, por ejemplo, son más eficientes con tareas de granularidad fina, mientras que las FPGAs pueden manejar mejor tareas de granularidad más gruesa.

4. **Administración de recursos:** La administración efectiva de recursos, incluyendo la asignación de tareas y la gestión de memoria, es crucial para mantener la eficiencia en sistemas heterogéneos.

Escalabilidad heterogénea

La escalabilidad se refiere a la capacidad de un sistema para mantener su rendimiento y eficiencia a medida que se incrementa el número de nodos o el tamaño de la carga de trabajo. En entornos heterogéneos, mantener la escalabilidad puede ser particularmente desafiante debido a las diferencias en las arquitecturas y capacidades de procesamiento.

Técnicas para mantener la escalabilidad

1. **Descomposición de tareas:** La descomposición eficiente de tareas en sub-tareas que puedan ser distribuidas entre diferentes tipos de procesadores es crucial para mantener la escalabilidad. Las técnicas de descomposición deben tener en cuenta las características específicas de los aceleradores utilizados.

2. **Algoritmos adaptativos:** Los algoritmos adaptativos que pueden ajustar dinámicamente la distribución de tareas y la asignación de recursos basándose en el estado actual del sistema pueden mejorar significativamente la escalabilidad. Estos algoritmos pueden redistribuir tareas entre procesadores y aceleradores para equilibrar la carga y minimizar el overhead.

3. **Modelo de programación heterogénea:** Un modelo de programación que soporte de manera efectiva la heterogeneidad de los recursos es esencial para la escalabilidad. Modelos como OpenCL y CUDA permiten la programación de aplicaciones para ejecutarse en diferentes tipos de aceleradores.

4. **Optimización de la comunicación:** Optimizar la comunicación entre los diferentes componentes del sistema es esencial para mantener la escalabilidad. Tecnologías como las redes de alta velocidad y las técnicas de comunicación asíncrona pueden reducir el overhead y mejorar la eficiencia.

Ejemplos de implementación

Aprendizaje profundo con GPUs y TPUs

En aplicaciones de aprendizaje profundo, la escalabilidad se logra mediante el uso de múltiples GPUs y TPUs. Las bibliotecas de aprendizaje profundo como TensorFlow y PyTorch soportan la distribución de tareas de entrenamiento y evaluación en múltiples aceleradores.

Ejemplo de uso de múltiples GPUs:

```
import tensorflow as tf

strategy = tf.distribute.MirroredStrategy()
```

```
with strategy.scope():
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(512, activation='relu',
input_shape=(784,)),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(10, activation='softmax')
    ])

    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

# Supongamos que `train_images` y `train_labels` son los datos de
entrenamiento
model.fit(train_images, train_labels, epochs=5)
```

Procesamiento de señales con FPGAs

En aplicaciones de procesamiento de señales, las FPGAs pueden ser utilizadas para implementar filtros digitales y otras operaciones de procesamiento de señales con baja latencia y alta eficiencia energética.

Ejemplo de implementación de un filtro digital en una FPGA:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOG

IC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity DigitalFilter is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           data_in : in STD_LOGIC_VECTOR (15 downto 0);
           data_out : out STD_LOGIC_VECTOR (15 downto 0));
end DigitalFilter;

architecture Behavioral of DigitalFilter is
    signal coeffs : array (0 to 4) of integer := (1, -2, 3, -2, 1);
    signal buffer : array (0 to 4) of integer;
begin
    process(clk, rst)
    begin
        if rst = '1' then
            buffer <= (others => 0);
            data_out <= (others => '0');
        elsif rising_edge(clk) then
            buffer(0) <= to_integer(unsigned(data_in));
```

```
            for i in 1 to 4 loop
                buffer(i) <= buffer(i - 1);
            end loop;
            data_out <= std_logic_vector(to_unsigned(buffer(0) *
coeffs(0) +
                                                    buffer(1) *
coeffs(1) +
                                                    buffer(2) *
coeffs(2) +
                                                    buffer(3) *
coeffs(3) +
                                                    buffer(4) *
coeffs(4), 16));
        end if;
    end process;
end Behavioral;
```

Computación en la nube con aceleradores heterogéneos

Los proveedores de servicios en la nube, como AWS, Google Cloud y Azure, ofrecen instancias con aceleradores heterogéneos que permiten a los desarrolladores desplegar y escalar aplicaciones de alto rendimiento utilizando una combinación de CPUs, GPUs y FPGAs.

Ejemplo de configuración de una instancia en la nube:

```
# AWS CLI para lanzar una instancia con GPU
aws ec2 run-instances --image-id ami-0abcdef1234567890 \
                      --instance-type p3.2xlarge \
                      --key-name MyKeyPair \
                      --security-groups my-sg
```

La integración de aceleradores hardware en sistemas heterogéneos requiere un enfoque cuidadoso para maximizar el speedup y la eficiencia, y mantener la escalabilidad en un entorno con recursos diversos y capacidades de procesamiento variadas.

Medición y optimización de rendimiento en computación paralela y distribuida

La medición y optimización del rendimiento son componentes cruciales para garantizar que los sistemas paralelos y distribuidos funcionen de manera eficiente y eficaz. A continuación, se presentan métodos y técnicas avanzadas para medir y optimizar el desempeño en estos sistemas.

Métodos de medición

1. **Profiling:**

   El profiling es una técnica utilizada para identificar las partes del código que consumen más tiempo y recursos. Herramientas como gprof, perf, y Intel VTune Amplifier son comúnmente utilizadas para este propósito. Estas herramientas

proporcionan un desglose detallado del tiempo de ejecución de funciones específicas, permitiendo a los desarrolladores identificar cuellos de botella en el rendimiento.

Ejemplo de uso de gprof:

```
gcc -pg -o my_program my_program.c
./my_program
gprof my_program gmon.out > analysis.txt
```

2. **Benchmarks:** Los benchmarks son pruebas estándar que permiten comparar el rendimiento de diferentes programas o sistemas bajo condiciones similares. Los benchmarks como SPEC MPI2007, LINPACK, y NAS Parallel Benchmarks son utilizados para evaluar el rendimiento de aplicaciones paralelas y distribuidas. Estos benchmarks proporcionan métricas comparativas que pueden ser utilizadas para evaluar la eficiencia y escalabilidad de los sistemas.

Optimización

1. **Reducción de la contención de recursos:** La contención de recursos ocurre cuando múltiples procesadores intentan acceder a un recurso compartido simultáneamente, causando retrasos. Para minimizar este problema, se pueden emplear técnicas como el uso de algoritmos de locking más eficientes (e.g., spinlocks, mutexes de bajo coste) y el diseño de algoritmos sin bloqueos (lock-free).

   Ejemplo de uso de mutexes en C++:

```
std::mutex mtx;
void thread_function() {
    std::lock_guard<std::mutex> guard(mtx);
    // Código que accede a recursos compartidos
}
```

2. **Mejora de la localidad de datos:** La localidad de datos se refiere a la proximidad de los datos en memoria. Mejorar la localidad de datos puede reducir significativamente la latencia de acceso a memoria. Técnicas como el bloqueado (blocking) y el uso de estructuras de datos que mejoren la cacheabilidad (e.g., arrays contiguos en lugar de listas enlazadas) son fundamentales.

   Ejemplo de bloqueado en multiplicación de matrices:

```
void blocked_matrix_multiply(int N, double **A, double **B,
double **C, int blockSize) {
    for (int ii = 0; ii < N; ii += blockSize) {
        for (int jj = 0; jj < N; jj += blockSize) {
            for (int kk = 0; kk < N; kk += blockSize) {
                for (int i = ii; i < std::min(ii + blockSize, N);
++i) {
```

```
                        for (int j = jj; j < std::min(jj + blockSize,
    N); ++j) {
                            for (int k = kk; k < std::min(kk +
    blockSize, N); ++k) {
                                C[i][j] += A[i][k] * B[k][j];
                            }
                        }
                    }
                }
            }
        }
    }
```

3. **Reducción de la sobrecarga de comunicación:** La comunicación entre procesadores puede ser un cuello de botella significativo en sistemas paralelos. Minimizar la cantidad de datos transferidos y utilizar métodos eficientes de comunicación, como la comunicación asincrónica y el agrupamiento de mensajes, puede mejorar el rendimiento.

   Ejemplo de uso de MPI para comunicación asincrónica:

```
    MPI_Request request;
    MPI_Isend(data, count, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD,
    &request);
    MPI_Wait(&request, MPI_STATUS_IGNORE);
```

Ejemplo: Optimización de un algoritmo de multiplicación de matrices

La multiplicación de matrices es una operación común en muchas aplicaciones científicas y de ingeniería. A continuación, se presenta un análisis detallado de cómo se puede optimizar este algoritmo tanto en su implementación secuencial como en su versión paralela.

Implementación secuencial:

```
def sequential_matrix_multiply(A, B):
    N = len(A)
    C = [[0] * N for _ in range(N)]
    for i in range(N):
        for j in range(N):
            for k in range(N):
                C[i][j] += A[i][k] * B[k][j]
    return C
```

Implementación paralela con multiprocessing:

```
import numpy as np
from multiprocessing import Pool
```

```
def parallel_matrix_multiply_worker(args):
    A, B, i, N = args
    C_row = np.zeros(N)
    for j in range(N):
        for k in range(N):
            C_row[j] += A[i][k] * B[k][j]
    return C_row

def parallel_matrix_multiply(A, B):
    N = len(A)
    C = np.zeros((N, N))
    with Pool() as pool:
        args = [(A, B, i, N) for i in range(N)]
        C_rows = pool.map(parallel_matrix_multiply_worker, args)
    for i in range(N):
        C[i] = C_rows[i]
    return C

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

Optimización adicional: uso de bloqueado: El uso de técnicas de bloqueado (blocking) puede mejorar la localidad de datos y reducir el acceso a la memoria principal, optimizando así el rendimiento del algoritmo.

Implementación de bloqueado:

```
def blocked_matrix_multiply(A, B, blockSize):
    N = len(A)
    C = [[0] * N for _ in range(N)]
    for ii in range(0, N, blockSize):
        for jj in range(0, N, blockSize):
            for kk in range(0, N, blockSize):
                for i in range(ii, min(ii + blockSize, N)):
                    for j in range(jj, min(jj + blockSize, N)):
                        for k in range(kk, min(kk + blockSize, N)):
                            C[i][j] += A[i][k] * B[k][j]
    return C

if __name__ == "__main__":
    N = 1000
    blockSize = 50
    A = np.random.rand(N, N)
    B = np.random.rand(N, N)
    C = blocked_matrix_multiply(A, B, blockSize)
```

Técnicas de balanceo de carga

El balanceo de carga es una técnica fundamental en la computación distribuida y en la administración de sistemas para mejorar la eficiencia y el rendimiento de los sistemas de múltiples nodos. A medida que las aplicaciones se vuelven más complejas y demandan mayor capacidad de procesamiento, el balanceo de carga avanzado se convierte en una necesidad crítica.

Balanceo de carga estático vs. dinámico

El balanceo de carga se puede clasificar en estático y dinámico. En el balanceo de carga estático, las tareas se distribuyen entre los nodos antes de la ejecución del programa y no cambian durante el tiempo de ejecución. Este enfoque es adecuado para sistemas con cargas de trabajo predecibles. Sin embargo, no es ideal para entornos donde las cargas de trabajo son variables e impredecibles.

El balanceo de carga dinámico, en cambio, ajusta la distribución de las tareas durante el tiempo de ejecución, basándose en el estado actual del sistema. Este enfoque es más adecuado para entornos dinámicos y heterogéneos donde la carga de trabajo puede variar significativamente.

Particionamiento de datos dinámico

El particionamiento de datos dinámico es una técnica avanzada de balanceo de carga que ajusta la distribución de datos entre nodos de acuerdo con las variaciones en la carga de trabajo y el estado del sistema. Esta técnica es particularmente útil en aplicaciones donde la carga de trabajo puede ser altamente irregular y cambiante.

1. **Hashing consistente:** El hashing consistente es una técnica utilizada para distribuir datos de manera uniforme entre nodos. A diferencia del hashing tradicional, el hashing consistente minimiza los cambios en la asignación de datos cuando se añaden o eliminan nodos. Esto es crucial para mantener el equilibrio de carga en sistemas distribuidos.

   Implementación: En un sistema de hashing consistente, cada nodo y cada clave se asignan a puntos en un anillo hash. Las claves son asignadas al nodo más cercano en el anillo, y cuando un nodo se añade o se elimina, solo un subconjunto de las claves se reasigna, minimizando la redistribución.

2. **Reparto basado en la carga:** En esta técnica, los datos se redistribuyen dinámicamente entre nodos en función de su carga actual. Los nodos sobrecargados pueden transferir parte de sus datos a nodos menos cargados para equilibrar la carga de trabajo.

   Ejemplo en MapReduce: En un sistema MapReduce, los datos de entrada se dividen en bloques, y las tareas de mapeo se asignan a nodos basándose en su proximidad a los datos. Durante la fase de reducción, los datos pueden redistribuirse para equilibrar la carga entre los nodos de reducción.

3. **Descomposición dinámica:** La descomposición dinámica ajusta la granularidad de las tareas basándose en la carga de trabajo actual. Por ejemplo, en un sistema de

simulación de partículas, las partículas pueden agruparse en regiones de diferente tamaño dependiendo de la densidad de partículas y la carga de procesamiento requerida.

Aplicación en simulación: En simulaciones de dinámica molecular, las regiones del espacio de simulación con alta densidad de partículas pueden subdividirse en celdas más pequeñas para distribuir la carga de procesamiento de manera más equitativa entre los nodos.

Algoritmos adaptativos

Los algoritmos adaptativos son una clase avanzada de técnicas de balanceo de carga que ajustan dinámicamente la asignación de tareas basándose en el rendimiento y el estado del sistema en tiempo real. Estos algoritmos pueden mejorar significativamente la eficiencia y la utilización de recursos en sistemas distribuidos.

1. **Algoritmos de rebalanceo:** Los algoritmos de rebalanceo ajustan la distribución de tareas durante la ejecución para mantener un equilibrio óptimo. Estos algoritmos monitorean continuamente el rendimiento del sistema y redistribuyen las tareas según sea necesario.

   Algoritmo de rebalanceo en gráficos: En la renderización de gráficos distribuidos, los nodos que se quedan atrás en el procesamiento de cuadros pueden transferir parte de su carga a nodos menos ocupados para mantener una tasa de cuadros uniforme.

2. **Algoritmos de aprendizaje por refuerzo:** Los algoritmos de aprendizaje por refuerzo utilizan técnicas de inteligencia artificial para aprender y adaptarse a las condiciones cambiantes del sistema. Estos algoritmos pueden mejorar el rendimiento a largo plazo mediante la optimización continua de la distribución de tareas.

   Aplicación en sistemas en tiempo real: En sistemas de control en tiempo real, los algoritmos de aprendizaje por refuerzo pueden ajustar la asignación de tareas en función de la retroalimentación del entorno, optimizando así el rendimiento y la eficiencia energética.

3. **Algoritmos de programación lineal:** Los algoritmos de programación lineal resuelven problemas de asignación de tareas mediante la optimización matemática. Estos algoritmos pueden encontrar la distribución óptima de tareas que minimiza el tiempo de ejecución total o maximiza la utilización de recursos.

   Ejemplo en computación en la nube: En entornos de computación en la nube, los algoritmos de programación lineal pueden optimizar la asignación de máquinas virtuales a las cargas de trabajo para minimizar el coste y maximizar el rendimiento.

4. **Algoritmos basados en el tiempo de ejecución:** Estos algoritmos ajustan la asignación de tareas basándose en el tiempo de ejecución observado de las tareas. Por ejemplo, en sistemas de renderización distribuida, las tareas de renderización

que tardan más tiempo en completarse pueden subdividirse y redistribuirse para equilibrar la carga.

Uso en renderización distribuida: En la renderización de películas CGI, las escenas complejas pueden dividirse en subescenas más pequeñas y asignarse a diferentes nodos, ajustando dinámicamente la carga en función del tiempo de renderización observado.

5. **Algoritmos genéticos:** Los algoritmos genéticos utilizan técnicas evolutivas para encontrar soluciones óptimas al problema del balanceo de carga. Estos algoritmos generan una población de posibles soluciones y las evolucionan mediante selección, cruce y mutación para encontrar la mejor distribución de tareas.

   Aplicación en redes neuronales distribuidas: En el entrenamiento distribuido de redes neuronales, los algoritmos genéticos pueden optimizar la distribución de datos y tareas entre los nodos para acelerar el tiempo de entrenamiento y mejorar la precisión del modelo.

6. **Modelos de predicción basados en el comportamiento:** Estos modelos utilizan datos históricos para predecir la carga futura y ajustar la distribución de tareas en consecuencia. Los modelos de predicción pueden basarse en técnicas estadísticas, machine learning o análisis de series temporales.

   Aplicación en gestión de infraestructura TI: En la gestión de infraestructuras TI, los modelos de predicción pueden anticipar picos de carga y redistribuir tareas preventivamente para evitar sobrecargas y garantizar un rendimiento óptimo.

Técnicas de minimizando el overhead

Minimizar el overhead es crucial para maximizar la eficiencia del balanceo de carga. El overhead puede provenir de la comunicación entre nodos, la sincronización y la gestión de tareas.

1. **Comunicación asincrónica:** La comunicación asincrónica permite a los nodos continuar procesando mientras esperan respuestas de otros nodos, reduciendo el tiempo de inactividad. Tecnologías como MPI (Message Passing Interface) soportan la comunicación asincrónica.

   Ejemplo con MPI:

```
MPI_Request request;
MPI_Isend(data, count, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD,
&request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
```

2. **Reducción de la frecuencia de sincronización:** Disminuir la frecuencia de sincronización entre nodos puede reducir el overhead de comunicación. Esto se puede lograr mediante técnicas como la relajación de la consistencia y el uso de modelos de consistencia eventual.

Uso en sistemas de base de datos distribuidas: En bases de datos distribuidas, la consistencia eventual permite que las actualizaciones se propaguen de manera asíncrona, reduciendo la necesidad de sincronización frecuente y mejorando el rendimiento.

3. **Agregación de mensajes:** La agregación de mensajes reduce el número de comunicaciones al combinar múltiples mensajes en uno solo. Esto es particularmente útil en sistemas donde el overhead de comunicación es alto.

   Aplicación en redes de sensores: En redes de sensores, los datos de múltiples sensores pueden agregarse en un solo mensaje antes de ser transmitidos, reduciendo el número de transmisiones y ahorrando energía.

4. **Planificación de tareas basada en prioridades:** Asignar prioridades a las tareas y planificarlas en consecuencia puede mejorar la eficiencia del sistema. Las tareas de alta prioridad pueden ejecutarse primero, reduciendo el tiempo de espera y mejorando el rendimiento general.

   Ejemplo en sistemas de tiempo real: En sistemas de control en tiempo real, las tareas críticas pueden asignarse una mayor prioridad para garantizar que se completen a tiempo, mientras que las tareas menos críticas pueden planificarse para ejecutarse en segundo plano.

El balanceo de carga es una técnica crucial en la computación paralela y distribuida que busca optimizar el uso de los recursos y maximizar el rendimiento del sistema.

Ejercicios

1 . Investiga y analiza un proyecto de investigación o una aplicación industrial donde se haya aplicado la Ley de Amdahl para mejorar el rendimiento del sistema.

Tareas:

- Selecciona un caso real de un proyecto de investigación o una aplicación industrial (e.g., simulaciones meteorológicas, procesamiento de imágenes médicas).
- Investiga cómo se aplicó la Ley de Amdahl para identificar las partes del sistema que podían ser paralelizadas.
- Analiza los resultados obtenidos en términos de speedup y eficiencia.
- Discute las limitaciones encontradas y cómo se abordaron.

Preguntas de discusión:

- ¿Qué fracción del código se pudo paralelizar P y cual es la fracción secuencial $(1-P)$.
- ¿Cuál fue el speedup teórico máximo según la Ley de Amdahl?
- ¿Cómo se comparan los resultados reales con los teóricos?
- ¿Qué técnicas se utilizaron para minimizar la fracción secuencial?

2 . Investiga y analiza un proyecto de investigación o una aplicación industrial donde se haya aplicado la Ley de Gustafson para mejorar el rendimiento del sistema.

Tareas:

- Selecciona un caso real de un proyecto de investigación o una aplicación industrial (e.g., renderización gráfica, análisis de grandes volúmenes de datos).
- Investiga cómo se aplicó la Ley de Gustafson para escalar la aplicación y mejorar el rendimiento.
- Analiza los resultados obtenidos en términos de speedup y eficiencia.
- Discute las ventajas y desventajas encontradas al aplicar esta ley.

Preguntas de discusión:

- ¿Cómo se determinó la fracción paralelizable (P) del código?
- ¿Cuál fue el speedup obtenido con un número creciente de procesadores?
- ¿Cómo se comparan los resultados reales con las predicciones de la Ley de Gustafson?
- ¿Qué mejoras en la eficiencia se observaron al escalar el tamaño del problema?

3 . Comparación práctica de las leyes de Amdahl y Gustafson

Implementa un experimento comparativo para observar cómo se comportan la Ley de Amdahl y la Ley de Gustafson en diferentes escenarios y tamaños de problema.

Tareas:

- Diseña un experimento utilizando una aplicación sencilla (por ejemplo, multiplicación de matrices).
- Implementa la versión secuencial y paralela del algoritmo.
- Mide el tiempo de ejecución para diferentes fracciones paralelizables (P) y diferentes números de procesadores.
- Calcula el speedup teórico y comparar con los resultados reales.

```
import numpy as np
import time
from multiprocessing import Pool

def matrix_multiply(A, B, start_row, end_row):
    C = np.zeros((end_row - start_row, B.shape[1]))
    for i in range(start_row, end_row):
        for j in range(B.shape[1]):
            C[i - start_row][j] = np.dot(A[i, :], B[:, j])
    return C

def parallel_matrix_multiply(A, B, num_workers):
    rows_per_worker = A.shape[0] // num_workers
    args = [(A, B, i * rows_per_worker, (i + 1) * rows_per_worker) for
i in range(num_workers)]

    with Pool(num_workers) as pool:
        results = pool.starmap(matrix_multiply, args)

    return np.vstack(results)
```

```
N = 1024
A = np.random.rand(N, N)
B = np.random.rand(N, N)

# Ejecución secuencial
start_time = time.time()
C_seq = matrix_multiply(A, B, 0, N)
end_time = time.time()
print(f"Tiempo secuencial: {end_time - start_time} segundos")

# Ejecución paralela
num_workers = 4
start_time = time.time()
C_par = parallel_matrix_multiply(A, B, num_workers)
end_time = time.time()
print(f"Tiempo paralelo con {num_workers} trabajadores: {end_time -
start_time} segundos")
```

Preguntas de discusión:

- ¿Cómo se comportan las dos leyes con diferentes fracciones paralelizables?
- ¿En qué escenarios la Ley de Gustafson ofrece mejores predicciones que la Ley de Amdahl?
- ¿Qué limitaciones se observan en ambos casos?

4 . Explora cómo la comunicación entre procesadores afecta el rendimiento y cómo se puede minimizar este overhead.

Tareas:

- Implementa un algoritmo de suma de vectores que distribuya los datos entre múltiples procesadores.
- Mide el tiempo de ejecución total y el tiempo dedicado a la comunicación entre procesadores.
- Experimenta con diferentes tamaños de datos y números de procesadores. . Analiza} el overhead de comunicación y proponer técnicas para minimizarlo (por ejemplo, comunicación asíncrona, agrupamiento de mensajes).

```
from mpi4py import MPI
import numpy as np

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

N = 1024
local_N = N // size

if rank == 0:
```

```
    vector = np.random.rand(N)
else:
    vector = None

local_vector = np.zeros(local_N)
comm.Scatter(vector, local_vector, root=0)

local_sum = np.sum(local_vector)
total_sum = comm.reduce(local_sum, op=MPI.SUM, root=0)

if rank == 0:
    print(f"Suma total: {total_sum}")
```

Preguntas de discusión:

¿Cómo afecta el overhead de comunicación al rendimiento global del algoritmo? ¿Qué técnicas se pueden utilizar para reducir este overhead? ¿Cómo cambia el overhead con el aumento del número de procesadores?

5 . Estudia la latencia y el ancho de banda en sistemas distribuidos y cómo influyen en la eficiencia del paralelismo.

Tareas:

- Implementa un algoritmo de comunicación punto a punto (por ejemplo, intercambio de mensajes) utilizando MPI.
- Mide la latencia y el ancho de banda para diferentes tamaños de mensajes.
- Analiza cómo estos parámetros afectan la eficiencia de un algoritmo paralelo.
- Propone mejoras para optimizar la comunicación en sistemas distribuidos.

```
from mpi4py import MPI
import numpy as np
import time

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

message_size = 1024 * 1024  # 1 MB
message = np.random.rand(message_size) if rank == 0 else
np.empty(message_size)

if rank == 0:
    start_time = time.time()
    comm.Send([message, MPI.DOUBLE], dest=1, tag=0)
    comm.Recv([message, MPI.DOUBLE], source=1, tag=1)
    end_time = time.time()
    print(f"Latencia: {end_time - start_time} segundos")
else:
    comm.Recv([message, MPI.DOUBLE], source=0, tag=0)
    comm.Send([message, MPI.DOUBLE], dest=0, tag=1)
```

6 . Implementa un algoritmo de multiplicación de matrices utilizando tanto CPU como GPU y analizar el speedup obtenido al variar el tamaño de las matrices.

Tareas:

1. Implementa la multiplicación de matrices en Python usando NumPy para la versión en CPU.
2. Implementa la multiplicación de matrices usando CUDA con PyCUDA para la versión en GPU.
3. Mide los tiempos de ejecución para diferentes tamaños de matrices (e.g., 256x256, 512x512, 1024x1024).
4. Calcula el speedup obtenido al usar la GPU en comparación con la CPU.
5. Analiza el impacto del tamaño de la matriz en el speedup y discutir los resultados.

Código base:

CPU:

```
import numpy as np
import time

def cpu_matrix_multiply(A, B):
    return np.dot(A, B)

N = 1024
A = np.random.rand(N, N).astype(np.float32)
B = np.random.rand(N, N).astype(np.float32)

start_time = time.time()
C = cpu_matrix_multiply(A, B)
end_time = time.time()

print(f"Tiempo de CPU: {end_time - start_time} segundos")
```

GPU:

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy as np
import time

N = 1024
A = np.random.rand(N, N).astype(np.float32)
B = np.random.rand(N, N).astype(np.float32)
C = np.zeros((N, N)).astype(np.float32)

A_gpu = cuda.mem_alloc(A.nbytes)
B_gpu = cuda.mem_alloc(B.nbytes)
C_gpu = cuda.mem_alloc(C.nbytes)
```

```
cuda.memcpy_htod(A_gpu, A)
cuda.memcpy_htod(B_gpu, B)

mod = SourceModule("""
__global__ void matrixMul(float *A, float *B, float *C, int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0;
    for (int k = 0; k < N; k++) {
        sum += A[row * N + k] * B[k * N + col];
    }
    C[row * N + col] = sum;
}
""")

matrixMul = mod.get_function("matrixMul")

block_size = 16
grid_size = (N + block_size - 1) // block_size

start_time = time.time()
matrixMul(A_gpu, B_gpu, C_gpu, np.int32(N), block=(block_size,
block_size, 1), grid=(grid_size, grid_size))
cuda.memcpy_dtoh(C, C_gpu)
end_time = time.time()

print(f" Tiempo de GPU: {end_time - start_time} segundos")

## Tus respuestas
```

7 . Implementa un filtro digital simple en una FPGA y analizar el rendimiento en comparación con una implementación en CPU.

Tareas:

1. Diseñar e implementar un filtro FIR en VHDL.
2. Implementar el mismo filtro en Python para ejecutarlo en una CPU.
3. Configurar y sintetizar el diseño en una FPGA.
4. Medir los tiempos de ejecución y el consumo de energía en ambos casos.
5. Comparar los resultados y discutir las ventajas y desventajas de cada enfoque.

Código base en VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity DigitalFilter is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           data_in : in STD_LOGIC_VECTOR (15 downto 0);
           data_out : out STD_LOGIC_VECTOR (15 downto 0));
end DigitalFilter;

architecture Behavioral of DigitalFilter is
    signal coeffs : array (0 to 4) of integer := (1, -2, 3, -2, 1);
    signal buffer : array (0 to 4) of integer;
begin
    process(clk, rst)
    begin
        if rst = '1' then
            buffer <= (others => 0);
            data_out <= (others => '0');
        elsif rising_edge(clk) then
            buffer(0) <= to_integer(unsigned(data_in));
            for i in 1 to 4 loop
                buffer(i) <= buffer(i - 1);
            end loop;
            data_out <= std_logic_vector(to_unsigned(buffer(0) *
coeffs(0) +
                                                     buffer(1) *
coeffs(1) +
                                                     buffer(2) *
coeffs(2) +
                                                     buffer(3) *
coeffs(3) +
                                                     buffer(4) *
coeffs(4), 16));
        end if;
    end process;
end Behavioral;

## Tus respuestas
```

8 . Implementa un algoritmo de simulación distribuida que utilice tanto CPUs como GPUs, y optimizar la comunicación entre estos componentes para minimizar el overhead.

Tareas:

1. Diseña un algoritmo de simulación que divida las tareas entre CPUs y GPUs.
2. Implementa el algoritmo utilizando PyCUDA para la parte de GPU y MPI para la comunicación entre nodos.
3. Mide los tiempos de comunicación y procesamiento por separado.
4. Optimiza la comunicación utilizando técnicas como el batching de mensajes y la comunicación asíncrona.
5. Analiza el impacto de las optimizaciones en el rendimiento general del sistema.

Código base:

```
from mpi4py import MPI
import numpy as np
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule

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

N = 1024
A = np.random.rand(N, N).astype(np.float32)
B = np.random.rand(N, N).astype(np.float32)
C = np.zeros((N, N)).astype(np.float32)

A_gpu = cuda.mem_alloc(A.nbytes)
B_gpu = cuda.mem_alloc(B.nbytes)
C_gpu = cuda.mem_alloc(C.nbytes)

cuda.memcpy_htod(A_gpu, A)
cuda.memcpy_htod(B_gpu, B)

mod = SourceModule("""
__global__ void matrixMul(float *A, float *B, float *C, int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0;
    for (int k = 0; k < N; k++) {
        sum += A[row * N + k] * B[k * N + col];
    }
    C[row * N + col] = sum;
}
""")

matrixMul = mod.get_function("matrixMul")

block_size = 16
grid_size = (N + block_size - 1) // block_size

# Split the work among nodes
if rank == 0:
    start_row = 0
    end_row = N // size
else:
    start_row = rank * (N // size)
    end_row = (rank + 1) * (N // size)
```

```
start_time = MPI.Wtime()
matrixMul(A_gpu, B_gpu, C_gpu, np.int32(N), block=(block_size,
block_size, 1), grid=(grid_size, grid_size))
cuda.memcpy_dtoh(C[start_row:end_row], C_gpu[start_row:end_row])
comm.Barrier()
end_time = MPI.Wtime()

if rank == 0:
    print(f"Tiempo total: {end_time - start_time}en segundos")

## Tu respuesta
```

9 . Diseña y ejecuta un experimento para evaluar la escalabilidad de un sistema heterogéneo que utilice una combinación de CPUs, GPUs y FPGAs.

Tareas:

1. Selecciona una aplicación adecuada (e.g., simulación de dinámica de fluidos, procesamiento de imágenes).
2. Implementa la aplicación utilizando diferentes aceleradores (CPUs, GPUs y FPGAs).
3. Configura un entorno de pruebas con una combinación de estos aceleradores.
4. Ejecuta la aplicación con diferentes tamaños de problema y número de nodos, y medir el rendimiento y la eficiencia.
5. Analiza los resultados para identificar cuellos de botella y discutir la escalabilidad del sistema.

Guía de implementación:

Configuración del entorno de pruebas:

1. Configura un cluster de computación con nodos que tengan GPUs y FPGAs.
2. Despliega el entorno de pruebas utilizando un gestor de recursos como Slurm.

Ejemplo de script de Slurm:

```
#!/bin/bash
#SBATCH --job-name=heterogeneous_test
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=1
#SBATCH --gres=gpu:2,fpga:1
#SBATCH --time=02:00:00
#SBATCH --output

=output_%j.txt

module load cuda
module load intel_fpga

srun --mpi=pmi2 python simulation.py
```

```
## Tus respuestas
```

10 .[Opcional]Desarrolla un modelo matemático para predecir el speedup y la eficiencia en un sistema heterogéneo basado en las características de los componentes del sistema y la carga de trabajo.

Tareas:

1. Revisa la literatura sobre modelos de speedup y eficiencia, como las Leyes de Amdahl y Gustafson.
2. Desarrolla un modelo que tenga en cuenta la heterogeneidad del sistema, incluyendo CPUs, GPUs y FPGAs.
3. Valida el modelo con datos experimentales obtenidos de la ejecución de aplicaciones en un sistema heterogéneo.
4. Compara los resultados del modelo con las mediciones reales y discutir cualquier discrepancia.
5. Ajusta el modelo según sea necesario para mejorar su precisión.

Ejemplo de fórmula:

$$S = \frac{T_s}{T_p} = \frac{T_s}{T_{cpu} + T_{gpu} + T_{fpga} + T_{com}}$$

Donde:

- T_s es el tiempo de ejecución en un sistema secuencial.
- T_p es el tiempo de ejecución en el sistema paralelo.
- T_{cpu}, T_{gpu}, T_{fpga} son los tiempos de ejecución en CPU, GPU y FPGA respectivamente.
- T_{com} es el tiempo de comunicación entre los componentes.

```
## Tus respuestas
```

11 . Utiliza herramientas de profiling y benchmarking para identificar cuellos de botella en el rendimiento de aplicaciones paralelas.

Tareas:

- Implementa un programa paralelo (por ejemplo multiplicación de matrices) en C o C++.
- Utiliza herramientas de profiling (e.g., gprof, perf) para identificar las partes del código que consumen más tiempo.
- Realiza benchmarking con diferentes tamaños de matrices y comparar los resultados.
- Optimiza el código basado en los resultados del profiling y benchmarking.
- Realiza los pasos anteriores usando python.

```
#include <iostream>
#include <vector>
#include <chrono>

using namespace std;
```

```
void matrix_multiply(const vector<vector<int>>& A, const
vector<vector<int>>& B, vector<vector<int>>& C, int N) {
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            C[i][j] = 0;
            for (int k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    int N = 1000;
    vector<vector<int>> A(N, vector<int>(N, 1));
    vector<vector<int>> B(N, vector<int>(N, 1));
    vector<vector<int>> C(N, vector<int>(N, 0));

    auto start = chrono::high_resolution_clock::now();
    matrix_multiply(A, B, C, N);
    auto end = chrono::high_resolution_clock::now();

    chrono::duration<double> diff = end - start;
    cout << "Tiempo de ejecución: " << diff.count() << " s" << endl;

    return 0;
}
```

Instrucciones de profiling:

- Compila con g++ -o matrix_mult matrix_mult.cpp -pg
- Ejecuta el programa: ./matrix_mult . Genera el reporte de profiling: gprof ./matrix_mult gmon.out > report.txt

```
## Tus respuestas.
```

12 . Implementa y evalua diferentes técnicas de balanceo de carga en aplicaciones paralelas.

Tareas:

- Implementa un algoritmo de simulación (por ejemplo el cálculo de pi) con balanceo de carga estático.
- Implementa el mismo algoritmo con balanceo de carga dinámico.
- Compara el rendimiento y la eficiencia de ambas implementaciones bajo diferentes cargas de trabajo.

```
# Ejemplo de balanceo estatico
from mpi4py import MPI
import numpy as np
```

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

N = 10000000
local_N = N // size

def compute_pi(local_N):
    count = 0
    for i in range(local_N):
        x = np.random.rand()
        y = np.random.rand()
        if x**2 + y**2 <= 1.0:
            count += 1
    return count

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

if rank == 0:
    pi = 4 * total_count / N
    print(f"Estimación de pi: {pi}")

##  Ejemplo para balanceo dinámico
from mpi4py import MPI
import numpy as np

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

N = 10000000
chunk_size = 1000

def compute_pi(chunk_size):
    count = 0
    for _ in range(chunk_size):
        x = np.random.rand()
        y = np.random.rand()
        if x**2 + y**2 <= 1.0:
            count += 1
    return count

total_count = 0

if rank == 0:
    tasks = N // chunk_size
    for i in range(1, size):
        comm.send(tasks, dest=i, tag=0)
```

```
    for i in range(tasks):
        local_count = comm.recv(source=MPI.ANY_SOURCE, tag=1)
        total_count += local_count
    for i in range(1, size):
        comm.send(0, dest=i, tag=0)  # Enviar señal de terminación
else:
    while True:
        tasks = comm.recv(source=0, tag=0)
        if tasks == 0:
            break
        local_count = compute_pi(chunk_size)
        comm.send(local_count, dest=0, tag=1)

if rank == 0:
    pi = 4 * total_count / N
    print(f"Estimación de pi: {pi}")


## Tus respuestas
```

13 . Mide y analiza la latencia y el ancho de banda de un sistema de comunicación paralelo.

Tareas:

- Implementa un programa para medir la latencia de comunicación punto a punto usando MPI.
- Implementa un programa para medir el ancho de banda de comunicación usando MPI.
- Analiza los resultados y discutir cómo la latencia y el ancho de banda afectan el rendimiento de aplicaciones paralelas.

```
# Ejemplo para latencia
from mpi4py import MPI
import time

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

if rank == 0:
    start_time = time.time()
    comm.send('ping', dest=1, tag=0)
    comm.recv(source=1, tag=1)
    end_time = time.time()
    print(f"Latencia: {end_time - start_time} segundos")

elif rank == 1:
    msg = comm.recv(source=0, tag=0)
    comm.send('pong', dest=0, tag=1)

# Ejemplo para ancho de banda
from mpi4py import MPI
import numpy as np
```

```
import time

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

N = 1000000  # Tamaño del mensaje
data = np.ones(N, dtype='float64')

if rank == 0:
    start_time = time.time()
    comm.Send([data, MPI.DOUBLE], dest=1, tag=0)
    end_time = time.time()
    print(f"Ancho de banda: {N * 8 / (end_time - start_time) / 1e6}
MB/s")

elif rank == 1:
    comm.Recv([data, MPI.DOUBLE], source=0, tag=0)



## Tus respuestas
```

14 . Evaluar el impacto del overhead de comunicación en el rendimiento de aplicaciones paralelas.

Tareas:

- Implementa un programa paralelo (e.g., suma de matrices) utilizando MPI.
- Mide el tiempo de ejecución con diferentes tamaños de mensajes y número de procesadores.
- Analiza el overhead de comunicación y su impacto en el rendimiento global.

```
from mpi4py import MPI
import numpy as np
import time

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

N = 10000  # Tamaño de las matrices
A = np.random.rand(N, N) if rank == 0 else None
B = np.random.rand(N, N) if rank == 0 else None
C = np.zeros((N, N)) if rank == 0 else None

if rank == 0:
    start_time = time.time()

# Distribuir las matrices a todos los procesos
A = comm.bcast(A, root=0)
B = comm.bcast(B, root=0)
```

```
# Suma de matrices en paralelo
rows_per_proc = N // size
local_A = A[rank*rows_per_proc:(rank+1)*rows_per_proc, :]
local_C = np.zeros_like(local_A)

for i in range(rows_per_proc):
    for j in range(N):
        local_C[i, j] = local_A[i, j] + B[i, j]

# Recopilar los resultados en el proceso raíz
comm.Gather(local_C, C, root=0)

if rank == 0:
    end_time = time.time()
    print(f"Tiempo total de ejecución: {end_time - start_time}
segundos")

## Tus respuestas
```