Artículo de Multiprocesadores

Implementación del método de Euler para calcular Pi

Bernardo Estrada (A01704320)

Tecnológico de Monterrey, Campus Querétaro

A01704320@itesm.mx

30 de noviembre de 2022

Resumen

En este artículo se discutirá cómo usar el multiprocesamiento para implementar el método de Euler para calcular pi usando hilos de Java, fork-join de Java, OpenMP, Intel TBB y CUDA. Exploraremos las ventajas y desventajas de cada enfoque y discutiremos los posibles aumentos de rendimiento.

Introducción

Calcular pi es un problema popular en el mundo de la matemática y la informática. El método de Euler para calcular pi, que establece que $pi^2/6 = 1/1^2 + 1/2^2 + 1/3^2 + ...$, es uno de los métodos más populares para hacerlo. Sin embargo, debido al gran número de cálculos requeridos, puede ser un proceso lento calcular pi de esta manera. Afortunadamente, aprovechando las tecnologías de multiprocesamiento como hilos de Java, fork-join de Java, OpenMP, Intel TBB y CUDA, es posible reducir significativamente el tiempo requerido para calcular pi usando el método de Euler. En este artículo, discutiremos cada uno de estos enfoques y exploraremos los aumentos de rendimiento que se pueden lograr.

Hilos de Java

Los hilos de Java son una forma sencilla de implementar multiprocesamiento en una aplicación de Java. Los hilos se pueden usar para dividir el trabajo de calcular pi en

múltiples tareas que pueden ser procesadas por un hilo separado. Esto puede resultar en aumentos de rendimiento significativos en comparación con un enfoque de un solo hilo. Sin embargo, es importante tener en cuenta que los hilos de Java solo se pueden usar para dividir el trabajo entre múltiples CPUs, no GPUs.

Fork-join de Java

Fork-join de Java es un enfoque más avanzado para el multiprocesamiento que se puede usar para dividir el trabajo entre CPUs y GPUs. El marco fork-join permite que las tareas se dividan en subtareas múltiples que luego pueden ser procesadas en paralelo por múltiples hilos. Esto puede resultar en aumentos de rendimiento significativos en comparación con un enfoque de un solo hilo.

OpenMP

OpenMP es un estándar abierto para el multiprocesamiento que se puede usar para dividir el trabajo entre CPUs y GPUs. OpenMP proporciona un conjunto de directivas del compilador que se pueden usar para dividir el trabajo en múltiples tareas que luego pueden ser procesadas en paralelo por múltiples hilos. Esto puede resultar en aumentos de rendimiento significativos en comparación con un enfoque de un solo hilo.

Intel TBB

Intel TBB es una biblioteca para multiprocesamiento que se puede usar para dividir el trabajo entre CPUs y GPUs. La biblioteca proporciona un conjunto de primitivas que se pueden usar para dividir el trabajo en múltiples tareas que luego pueden ser procesadas en paralelo por múltiples hilos. Esto puede resultar en aumentos de rendimiento significativos en comparación con un enfoque de un solo hilo.

CUDA

CUDA es una tecnología para el multiprocesamiento que se puede usar para dividir el trabajo entre GPUs. CUDA proporciona un conjunto de herramientas que se pueden usar para dividir el trabajo en múltiples tareas que luego pueden ser procesadas en paralelo por múltiples hilos. Esto puede resultar en aumentos de rendimiento significativos en comparación con un enfoque de un solo hilo.

Conclusión

En este artículo, discutimos cómo usar el multiprocesamiento para implementar el método de Euler para calcular pi. Exploramos las ventajas y desventajas de usar hilos de Java, fork-join de Java, OpenMP, Intel TBB y CUDA. Cada uno de estos enfoques puede resultar en aumentos de rendimiento significativos en comparación con un enfoque de un solo hilo. Es importante tener en cuenta que el mejor enfoque dependerá del uso específico.

Referencias

Pathik Pathik (2022). Recuperado el 1 de diciembre de 2022, desde https://www.youtube.com/watch?v=qh73-x5sRXI

GitHub - oneapi-src/oneTBB: oneAPI Threading Building Blocks (oneTBB). (2022). Recuperado el 1 de diciembre de 2022, desde https://github.com/oneapi-src/oneTBB

CUDA Toolkit Documentation. (2022). Recuperado el 1 de diciembre de 2022, desde https://docs.nvidia.com/cuda/

Apéndice

Hilos de Java

```
// ================================================================
//
// File: Threads_EulerPi.java
// Author: Bernardo Estrada
// Description: This file contains the code to calculate Pi
//              with Euler's method using Java's Threads.
//  To compile:
//   javac Threads_EulerPi.java
//  To run:
//   java Threads_EulerPi
//
// Copyright (c) 2022 by Tecnologico de Monterrey.
// All Rights Reserved. May be reproduced for any non-commercial
// purpose.
//
// ================================================================

public class Threads_EulerPi {
  private static final long MAX_N = 394_656_595L;
```

```
  private static final long N = 394_656_595L;
  private static final int THREADS = Runtime.getRuntime().availableProcessors();
  private static final long BLOCK = N / (long)THREADS;
  private static final long N2 = N * N;

  private static class EulerPi extends Thread {
    private long start, end, sum;

    public EulerPi(long start, long end) {
      this.start = start;
      this.end = end;
      this.sum = 0;
    }

    public double getSum() {
      return (double)sum / N2;
    }

    public long getLongSum() {
      return sum;
    }

    @Override
    public void run() {
      for (long i = start; i < end; i++) {
        if (i==0) continue;
        sum += N2 / (i * i);
      }
    }
  }

  public static void main(String args[]) {
    EulerPi threads[] = new EulerPi[THREADS];
    double pi;

    if (N > MAX_N) {
      System.out.printf("N must be <= %d %n", MAX_N);
      System.exit(-1);
    }

    for (int i = 0; i < THREADS; i++) {
      threads[i] = new EulerPi(i * BLOCK , (i + 1) * BLOCK);
      threads[i].start();
    }

    try {
      for (int i = 0; i < THREADS; i++) {
        threads[i].join();
      }
    } catch (InterruptedException ie) {
      System.out.printf("main: %s %n", ie);
    }
```

```
    System.out.printf("sum = %f %n", (double)(6 * sum(threads))/N2);
    pi = Math.sqrt((double)(6 * sum(threads))/N2);
    System.out.printf("pi = %.10f %n", pi);
  }

  private static long sum(EulerPi threads[]) {
    long sum = 0;

    for (int i = 0; i < THREADS; i++) {
      sum += threads[i].getLongSum();
    }
    return sum;
  }
}
```

Fork-join de Java

```
// ============================================================================
//
// File: ForkJoin_EulerPi.c
// Author: Bernardo Estrada
// Description: This file contains the code to calculate Pi
//              with Euler's method using Java's Fork-Join.
//  To compile:
//   javac ForkJoin_EulerPi.java
//  To run:
//   java ForkJoin_EulerPi
//
// Copyright (c) 2022 by Tecnologico de Monterrey.
// All Rights Reserved. May be reproduced for any non-commercial
// purpose.
//
// ============================================================================

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class ForkJoin_EulerPi {
  private static final long MAX_N = 394_656_595L;
  private static final long N = 394_656_595L;
  private static final int THREADS = Runtime.getRuntime().availableProcessors();
  private static final long BLOCK = N / (long)THREADS;
  private static final long N2 = N * N;

  private static class EulerPi extends RecursiveTask<Long> {
    private long start, end;

    public EulerPi(long start, long end) {
      this.start = start;
      this.end = end;
```

```
    }

    @Override
    protected Long compute() {
      long sum = 0;

      if ((end - start) <= BLOCK) {
        for (long i = start; i < end; i++) {
          if (i==0) continue;
          sum += N2 / (i * i);
        }
      } else {
        long mid = (start + end) / 2;
        EulerPi t1 = new EulerPi(start, mid);
        EulerPi t2 = new EulerPi(mid, end);
        t1.fork();
        t2.fork();
        sum = t1.join() + t2.join();
      }
      return sum;
    }
  }

  public static void main(String args[]) {
    EulerPi task;
    double pi;

    if (N > MAX_N) {
      System.out.printf("N must be <= %d %n", MAX_N);
      System.exit(-1);
    }

    task = new EulerPi(0, N);
    ForkJoinPool pool = new ForkJoinPool(THREADS);
    pi = Math.sqrt((double)(6 * pool.invoke(task))/N2);
    System.out.printf("pi = %.10f%n", pi);
  }
}
```

OpenMP

```
// ================================================================
//
// File: OpenMP_EulerPi.c
// Author: Bernardo Estrada
// Description: This file contains the code to calculate Pi
//              with Euler's method using OpenMP.
//  To compile:
//      gcc -o OpenMP_EulerPi OpenMP_EulerPi.c -lm -fopenmp
//  To run:
```

```
//      ./OpenMP_EulerPi
//
// Copyright (c) 2022 by Tecnologico de Monterrey.
// All Rights Reserved. May be reproduced for any non-commercial
// purpose.
//
// ================================================================

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

#define MAX_N 394656595L
#define N 394656595L

int main(int argc, char* argv[]) {
  const long N2 = N*N;
  double pi;
  long sum = 0;
  long i;

  if (N > MAX_N) {
    printf("N must be <= %ld (MAX_N)\n", MAX_N);
    exit(1);
  }

  #pragma omp parallel for reduction(+:sum)
  for (i = 0; i < N; i++) {
    if (i == 0) continue;
    sum += N2 / (i * i);
  }
  pi = sqrt((double)(6 * sum)/N2);
  printf("pi = %lf\n", pi);
  return 0;
}
```

Intel TBB

```
// ================================================================
//
// File: OpenMP_EulerPi.c
// Author: Bernardo Estrada
// Description: This file contains the code to calculate Pi
//              with Euler's method using Intel's TBB.
//  To compile:
//      g++ -o TBB_EulerPi TBB_EulerPi.cpp -ltbb
//  To run:
//      ./TBB_EulerPi
//
```

```
// Copyright (c) 2022 by Tecnologico de Monterrey.
// All Rights Reserved. May be reproduced for any non-commercial
// purpose.
//
// ================================================================

// implement euler's method to calculate pi using intel tbb

#include <iostream>
#include <iomanip>
#include <math.h>
#include <tbb/parallel_for.h>
#include <tbb/blocked_range.h>

using namespace std;
using namespace tbb;

const long MAX_N = 394656595L;
const long N = 394656595L;
const long N2 = N*N;

class EulerPi {
private:
  long *sum;
public:
  EulerPi(long *sum) : sum(sum) { }
  void operator()(const blocked_range<long> &r) const {
    long local_sum = 0.0;
    for (long i = r.begin(); i != r.end(); i++) {
      if (i != 0)
        local_sum += N2 / (i * i);
    }
    *sum += local_sum;
  }
};

double eulerPi() {
  long sum = 0;
  parallel_for(blocked_range<long>(0, N), EulerPi(&sum));
  return sum;
}

int main(int argc, char* argv[]) {
  double pi;
  long sum = eulerPi();

  pi = sqrt((double)(6 * sum)/N2);
  cout << "pi = " << setprecision(10) << pi << endl;
  return 0;
}
```

CUDA

```
// ================================================================
//
// File: CUDA_EulerPi.c
// Author: Bernardo Estrada
// Description: This file contains the code to calculate Pi
//              with Euler's method using Nvidia's CUDA.
//  To compile:
//      nvcc -o CUDA_EulerPi CUDA_EulerPi.cu
//  To run:
//      ./CUDA_EulerPi
//
// Copyright (c) 2022 by Tecnologico de Monterrey.
// All Rights Reserved. May be reproduced for any non-commercial
// purpose.
//
// ================================================================

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda.h>

#define N 1000000000

__global__ void eulerPi(double *pi, double *sum, int n) {
  int i, tid = threadIdx.x + blockIdx.x * blockDim.x;
  double h = 1.0 / n;

  for (i = tid; i < n; i += blockDim.x * gridDim.x) {
    sum[tid] += 4.0 / (1.0 + ((i + 0.5) * h) * ((i + 0.5) * h));
  }
  __syncthreads();

  if (tid == 0) {
    for (i = 0; i < blockDim.x * gridDim.x; i++) {
      *pi += sum[i];
    }
    *pi *= h;
  }
}

int main(int argc, char* argv[]) {
  double *pi, *sum, *d_pi, *d_sum;
  int i, blockSize, gridSize;

  // allocate memory
  pi = (double *) malloc(sizeof(double));
  sum = (double *) malloc(sizeof(double) * 1024);
  cudaMalloc((void **) &d_pi, sizeof(double));
  cudaMalloc((void **) &d_sum, sizeof(double) * 1024);
```

```
  // initialize variables
  *pi = 0;
  for (i = 0; i < 1024; i++) {
    sum[i] = 0;
  }

  // copy data to device
  cudaMemcpy(d_pi, pi, sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(d_sum, sum, sizeof(double) * 1024, cudaMemcpyHostToDevice);

  // calculate pi
  blockSize = 1024;
  gridSize = (N + blockSize - 1) / blockSize;
  eulerPi<<<gridSize, blockSize>>>(d_pi, d_sum, N);

  // copy data back to host
  cudaMemcpy(pi, d_pi, sizeof(double), cudaMemcpyDeviceToHost);

  // print results
  printf("pi = %lf (error = %lf) \n", *pi, fabs(*pi - M_PI));

  // free memory
  free(pi);
  free(sum);
  cudaFree(d_pi);
  cudaFree(d_sum);
  return 0;
}
```