Leyes de Amdahl y Gustafson-Barsis y Medición de Rendimiento en Programas de OpenMP

Christian Asch

October 2, 2024

Contents

1 Introducción al rendimiento paralelo y la escalabilidad **2**
- 1.1 Speedup . . . 2
- 1.2 Escalabilidad . . . 2
- 1.3 Eficiencia . . . 2
- 1.4 Partes secuenciales y paralelas de los programas . . . 2
- 1.5 Overhead paralelo . . . 2

2 Ley de Amdahl **2**
- 2.1 Introducción a la ley de Amdahl . . . 2
  - 2.1.1 Fórmula y derivación . . . 2
- 2.2 Aplicabilidad de la ley de Amdahl . . . 3
- 2.3 Ejemplo y graficación . . . 3

3 Ley de Gustafson **4**
- 3.1 Introducción a la ley de Gustafson . . . 4
  - 3.1.1 Fórmula y derivación . . . 4
- 3.2 Aplicabilidad de la ley de Gustafson . . . 4
- 3.3 Ejemplo y graficación . . . 4

4 Descanso (15 minutos) **5**

5 Cómo medir el tiempo de programas de OpenMP? **5**
- 5.1 Jerarquía de memoria . . . 5
- 5.2 Makefile y programa . . . 6
- 5.3 Medición del speedup y la eficiencia de un programa de OpenMP 6
- 5.4 Posibles problemas de rendimiento . . . 8

5.5 Strong scaling 9
5.6 Weak scaling 9
5.7 Práctica con multiplicación de matrices 9

6 Resumen y preguntas (15 minutos) 12

1 Introducción al rendimiento paralelo y la escalabilidad

1.1 Speedup

El speedup es una comparación relativa entre dos casos de ejecución.

1.2 Escalabilidad

La escalabilidad se refiere a la forma en que los programas responden a distintos niveles de recursos.

1.3 Eficiencia

Qué tan bien se usan los recursos para obtener el Speedup

1.4 Partes secuenciales y paralelas de los programas

1.5 Overhead paralelo

2 Ley de Amdahl

2.1 Introducción a la ley de Amdahl

2.1.1 Fórmula y derivación

$$S_n = \frac{t_1}{t_n}$$

$$t_1 = t_s + t_p = 1$$

$$t_n = t_s + \frac{t_p}{n} = 1 - t_p + \frac{t_p}{n}$$

$$S_n = \frac{1}{(1 - t_p) + \frac{t_p}{n}}$$

$$\lim_{n\to\infty} S_n = \frac{1}{1 - t_p}$$

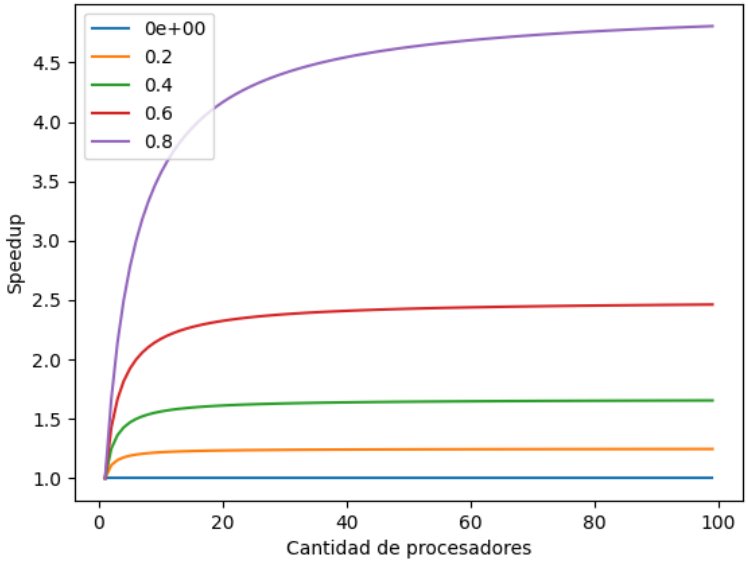
2.2 Aplicabilidad de la ley de Amdahl

2.3 Ejemplo y graficación

En este código graficamos la cantidad de procesadores contra el speedup obtenido para varios programas con distintos porcentajes de paralelismo.

```
import matplotlib.pyplot as plt
import numpy as np
def amdahl_speedup(p, n):
    return 1.0 / (1 - p + p/n)

name = "./amdahl.png"
procs = np.arange(1, 100)
parallel_portions = np.arange(0, 1, 0.2)
plt.xlabel("Cantidad de procesadores")
plt.ylabel("Speedup")
for par in parallel_portions:
    result = amdahl_speedup(par, procs)
    plt.plot(procs, result, label=f"{par:.1}")
plt.legend()
plt.savefig(name)
return name
```



3 Ley de Gustafson

3.1 Introducción a la ley de Gustafson

3.1.1 Fórmula y derivación

$$S_n = \frac{t_1}{t_n}$$

$$t_n = t_s + t_p = 1$$

$$t_1 = t_s + n \cdot t_p$$

$$S_n = 1 - t_p + n \cdot t_p$$

$$S_n = 1 + t_p \cdot (n - 1)$$

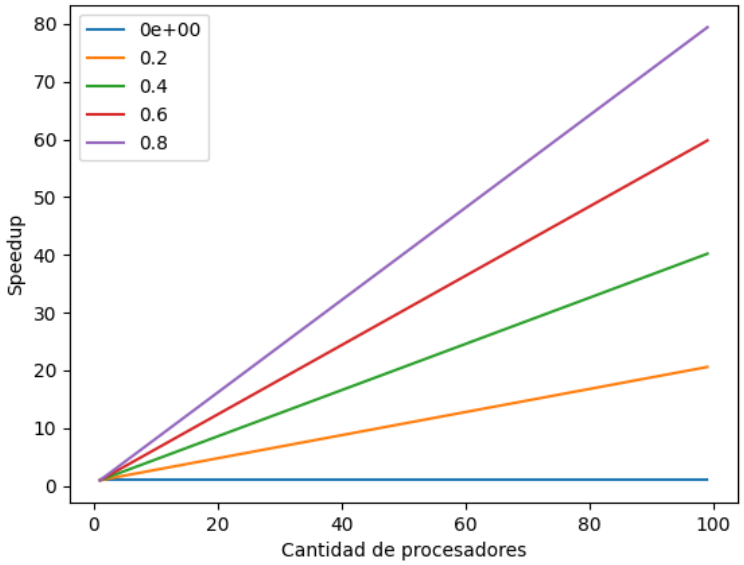
3.2 Aplicabilidad de la ley de Gustafson

3.3 Ejemplo y graficación

```
import matplotlib.pyplot as plt
import numpy as np
def gustafson_speedup(p, n):
```

```
    return 1 + p * (n - 1)

name = "./gustafson.png"
procs = np.arange(1, 100)
parallel_portions = np.arange(0, 1, 0.2)
plt.xlabel("Cantidad de procesadores")
plt.ylabel("Speedup")
for par in parallel_portions:
    result = gustafson_speedup(par, procs)
    plt.plot(procs, result, label=f"{par:.1}")
plt.legend()
plt.savefig(name)
return name
```



4 Descanso (15 minutos)

5 Cómo medir el tiempo de programas de OpenMP?

5.1 Jerarquía de memoria

- Concepto

- L1d, L1i, L2, L3 Cache
- Cache lines

Para realizar los siguientes ejemplos nos vamos a concentrar en la operación SAXPY, "Single precision A X plus Y". Esta operación es la siguiente:

$$\boldsymbol{z} = a\,\boldsymbol{x} + \boldsymbol{y}$$

5.2 Makefile y programa

Las primeras líneas del Makefile nos indican cuál es el compilador que utilizaremos, así como banderas necesarias para realizar la compilación. En este caso utilizamos `-fopenmp` para que el programa pueda encontrar las bibliotecas necesarias.

```
CC=gcc-14
FLAGS=-fopenmp -O3
```

Luego definimos los comandos de compilación. En este contexto, `sm` significa "shared memory".

```
all: saxpy_serial saxpy_sm

saxpy_serial: saxpy_serial.c
 ${CC} ${FLAGS} -o $@ $?

saxpy_sm: saxpy_sm.c
 ${CC} ${FLAGS} -o $@ $?
```

5.3 Medición del speedup y la eficiencia de un programa de OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>
void saxpy(float a, float * x, float * y, int elements)
{
  for(int i = 0; i < elements; ++i)
  {
```

```
    y[i] = a * x[i] + y[i];
  }
}
int main(int argc, char ** argv)
{
  double start, end;
  int total_elements = atoi(argv[1]);
  float a, *x, *y;
  a = 10.f;
  start = omp_get_wtime();
  x = malloc(sizeof(float) * total_elements);
  y = malloc(sizeof(float) * total_elements);

  for(int i = 0; i < total_elements; ++i)
  {
    x[i] = 1.f;
    y[i] = 2.3f;
  }
  end = omp_get_wtime() - start;
  printf("Init time: %f\n", end);
  start = omp_get_wtime();
  saxpy(a, x, y, total_elements);
  end = omp_get_wtime() - start;
  printf("Execution time: %f\n", end);
  free(x);
  free(y);
  return 0;
}


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>
void saxpy(float a, float * x, float * y, const int elements, const int chunk_size)
{
#pragma omp parallel for schedule(static, chunk_size) default(none) shared(a, x, y, chu
  for(int i = 0; i < elements; ++i)
  {
    y[i] = a * x[i] + y[i];
```

```
  }
}
int main(int argc, char ** argv)
{
  double start, end;
  int total_elements = atoi(argv[1]);
  int chunk_size = atoi(argv[2]);
  float a, *x, *y;
  a = 10.f;
  start = omp_get_wtime();
  x = malloc(sizeof(float) * total_elements);
  y = malloc(sizeof(float) * total_elements);

#pragma omp parallel for schedule(static, chunk_size) default(none) shared(total_eleme
  for(int i = 0; i < total_elements; ++i)
  {
    x[i] = 1.f;
    y[i] = 2.3f;
  }
  end = omp_get_wtime() - start;
  printf("Init time: %f\n", end);
  start = omp_get_wtime();
  saxpy(a, x, y, total_elements, chunk_size);
  end = omp_get_wtime() - start;
  printf("Execution time: %f\n", end);
  free(x);
  free(y);
  return 0;
}
```

5.4 Posibles problemas de rendimiento

- Overhead
- False sharing

5.5 Strong scaling

5.6 Weak scaling

5.7 Práctica con multiplicación de matrices

```
CC=gcc-14
FLAGS=-fopenmp -O3
all: matmul_serial

matmul_serial: matmul_serial.c
 ${CC} ${FLAGS} -o $@ $?
```

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

typedef struct {
  double * data;
  int height;
  int width;
} Matrix;

inline double get_element(const Matrix *mat, int i, int j);
inline void set_element(Matrix *mat, int i, int j, double val);

double get_element(const Matrix *mat, int i, int j)
{
  int index = mat->width * i + j;
  return mat->data[index];
}

void set_element(Matrix *mat, int i, int j, double val)
{
  int index = mat->width * i + j;
  mat->data[index] = val;
}

void free_matrix(Matrix *mat)
{
```

```
  free(mat->data);
  mat->data = NULL;

  free(mat);
  mat = NULL;
}

Matrix * alloc_matrix(int height, int width)
{
  int total_elements = height * width;
  Matrix *mat = malloc(sizeof(Matrix));
  mat->height = height;
  mat->width = width;
  mat->data = malloc(sizeof(double) * total_elements);
  return mat;
}

Matrix * create_n_matrix(int height, int width, int n)
{
  int total_elements = height * width;
  Matrix *mat = alloc_matrix(height, width);
  memset(mat->data, n, total_elements);
  return mat;
}

Matrix * create_rand_matrix(int height, int width)
{
  int total_elements = height * width;
  Matrix *mat = alloc_matrix(height, width);

  for(int i = 0; i < total_elements; ++i)
  {
    mat->data[i] = 1. + rand()/(RAND_MAX + 1.)/10;
  }

  return mat;
}

Matrix * matmul(const Matrix *matA, const Matrix *matB)
{
```

```
  Matrix * matC = create_n_matrix(matA->height, matB->width, 0);
  for(int i = 0; i < matA->height; ++i)
  {
    for(int j = 0; j < matB->width; ++j)
    {
      double value = .0;
      for(int k = 0; k < matA->width; ++k)
      {
        value += get_element(matA, i, k) * get_element(matB, k, j);
      }
      set_element(matC, i, j, value);
    }
  }
  return matC;
}

void print_matrix(const Matrix * mat)
{
  for(int i = 0; i < mat->height; ++i)
  {
    for(int j = 0; j < mat->width; ++j)
    {
      printf("%f ", get_element(mat, i, j));
    }
    printf("\n");
  }
}

int main()
{
  Matrix * matA = create_rand_matrix(3, 4);
  Matrix * matB = create_rand_matrix(4, 1);
  Matrix * matC = matmul(matA, matB);
  print_matrix(matC);
  free_matrix(matA);
  free_matrix(matB);
  free_matrix(matC);
  return 0;
}
```

6 Resumen y preguntas (15 minutos)