Parallel-Programming Task3

刘森元, 21307289

中山大学计算机学院

Codes on https://github.com/Myocardial-infarction-Jerry/Parallel-Programming/tree/main/Task3.

Project built by CMake.

```
> cd Task3
> cmake . && make
> ./PthreadMatMul
> ./PthreadVecSum
```

1 Environment

11th Gen Intel(R) Core(TM) i7-11700KF @ 3.60GHz

NVIDIA GeForce RTX 3080 Ti O12G

Windows Subsystem for Linux @ Ubuntu 22.04 LTS

2 Task

2.1 并行矩阵乘法

使用 Pthread 实现并行矩阵乘法, 并通过实验分析其性能.

> **Note**
>
> **输入:** m, n, k 三个整数, 每个整数的取值范围均为 $[128, 2048]$.
>
> **问题描述:** 随机生成 $m \times n$ 的矩阵 A 以及 $n \times k$ 的矩阵 B, 并对这两个矩阵进行矩阵乘法运算, 得到矩阵 C.
>
> **输出:** A, B, C 三个矩阵, 及矩阵计算所消耗的时间 t.
>
> **要求:**
>
> 1. 使用 Pthread 创建多线程实现并行矩阵乘法, 调整线程数量 (1-16) 及矩阵规模 (128-2048), 根据结果分析其并行性能 (包括但不限于, 时间、效率、可扩展性).
> 2. 选做: 可分析不同数据及任务划分方式的影响.

2.2 并行数组求和

使用 Pthread 实现并行数组求和, 并通过实验分析其性能.

> **Note**
>
> **输入:** 整数 n, 取值范围为 $[1\text{M}, 128\text{M}]$
>
> **问题描述:** 随机生成长度为 n 的整型数组 A, 计算其元素和 $s = \sum_{i=1}^{n} A_i$.

> **输出：** 数组 A, 元素和 s, 即求和计算所消耗的时间 t.
>
> **要求：**
>
> 1. 使用 Pthreads 实现并行数组求和, 调整线程数量 (1-16) 及矩阵规模 (128-2048), 根据结果分析其并行性能 (包括但不限于, 时间、效率、可扩展性).
> 2. 选做: 可分析不同聚合方式的影响.

3 Theory

3.1 并行矩阵乘法

由于矩阵的可分性, 使用并行通用矩阵乘法的朴素思想, 即将矩阵分割为 $size$ 等份, 在多个线程中进行计算, 最后由 master 核进行拼接.

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{bmatrix}$$

如上图的逐行计算, 对于 $m \times n$ 的矩阵 A, 可将其切分为若干个 $subM \times n$ 子矩阵分别与矩阵 B 相乘, 最后进行拼接, 其中 $subM = (m + size - 1)/size$, 保证均分.

3.2 并行数组求和

由于求和的可分性, 可将数组等分至 $size$ 个线程中分别求和, 最后进行统一求和.

4 Code

4.1 并行矩阵乘法

> **Caution**
>
> 源代码详见 PthreadMatMul.cpp

4.1.1 朴素矩阵乘法

```
1  void *matMulThread(void *arg) {
2      ThreadData *data = (ThreadData *)arg;
3      float *A = data->A;
4      float *B = data->B;
5      float *C = data->C;
6      int m = data->m;
7      int n = data->n;
8      int k = data->k;
9      int startRow = data->startRow;
10     int endRow = data->endRow;
```

```
11
12      // Perform matrix multiplication C = A * B for the assigned rows
13      for (int i = startRow; i < endRow; ++i) {
14          for (int j = 0; j < k; ++j) {
15              C[i * k + j] = 0;
16              for (int l = 0; l < n; ++l) {
17                  C[i * k + j] += A[i * n + l] * B[l * k + j];
18              }
19          }
20      }
21
22      pthread_exit(NULL);
23  }
```

4.1.2 进行矩阵划分

```
1   // Assign rows to each thread
2   int startRow = 0;
3   for (int i = 0; i < NUM_THREADS; ++i) {
4       int endRow = startRow + rowsPerThread;
5       if (i < remainingRows)
6           endRow++;
7
8       // Create thread data
9       ThreadData *data = new ThreadData;
10      data->A = A;
11      data->B = B;
12      data->C = C;
13      data->m = m;
14      data->n = n;
15      data->k = k;
16      data->startRow = startRow;
17      data->endRow = endRow;
18
19      // Create thread and pass thread data
20      int rc = pthread_create(&threads[i], &attr, matMulThread, (void *)data);
21      if (rc) {
22          std::cerr << "Error: Unable to create thread, return code: " << rc << std::endl;
23          exit(-1);
24      }
25
26      startRow = endRow;
27  }
```

4.2 并行数组求和

> **Caution**
>
> 源代码详见 PthreadVecSum.cpp

4.2.1 朴素数组求和

```
1   void *sumArrayThread(void *arg) {
2       ThreadData *data = (ThreadData *)arg;
```

```
    int *array = data->array;
    int size = data->size;
    int startIdx = data->startIdx;
    int endIdx = data->endIdx;
    int sum = 0;

    // Calculate the sum of the assigned portion of the array
    for (int i = startIdx; i < endIdx; ++i)
        sum += array[i];

    data->sum = sum;
    pthread_exit(NULL);
}
```

4.2.2 进行数组划分

```
// Assign elements to each thread
int startIdx = 0;
ThreadData threadData[NUM_THREADS];
for (int i = 0; i < NUM_THREADS; ++i) {
    int endIdx = startIdx + elementsPerThread;
    if (i < remainingElements)
        endIdx++;

    // Create thread data
    threadData[i].array = array;
    threadData[i].size = size;
    threadData[i].startIdx = startIdx;
    threadData[i].endIdx = endIdx;

    // Create thread and pass thread data
    int rc = pthread_create(&threads[i], &attr, sumArrayThread, (void *)&threadData[i]);
    if (rc) {
        std::cerr << "Error: Unable to create thread, return code: " << rc << std::endl;
        exit(-1);
    }

    startIdx = endIdx;
}
```

5 Result

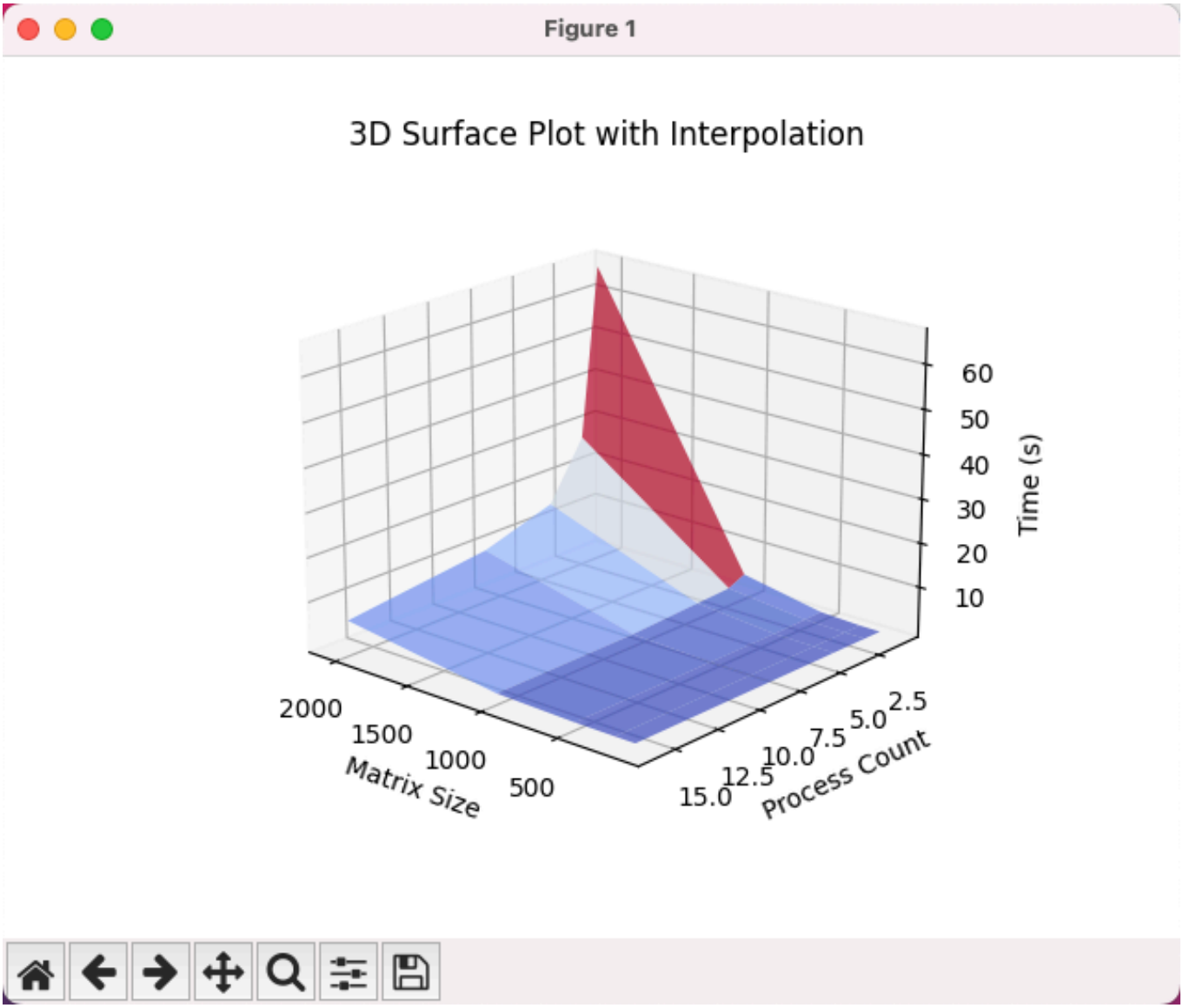
5.1 并行矩阵乘法

以 8 核心 512 矩阵为例

```
Mon  8 Apr - 17:55  ~/GitHub/Parallel-Programming/Task3  origin main 2●
chef@ChefMichelin-PC  ./PthreadMatMul
512 512 512
Calculating for m = 512, n = 512, k = 512
Running on 8 processes
matMul time: 0.113000 seconds
Verifying with Python script
The matrix multiplication is correct.
```

其中 `matMul time` 为并行矩阵乘法运行时间.

进程数量/矩阵规模 (s)	128	256	512	1024	2048
1	0.006114	0.048693	0.455058	3.876503	66.468744
2	0.003178	0.023870	0.241512	2.133491	27.681108
4	0.001889	0.012669	0.143153	1.716617	14.663887
8	0.001630	0.007008	0.112066	0.869376	9.596393
16	0.001113	0.006665	0.062423	0.572545	5.646903



- 随着进程数量的增加, 程序的运行时间在大多数情况下都有所减少.

- 当进程数量增加到 16 时, 对于规模为 2048 的矩阵, 运行时间并没有显著减少. 这是因为进程间的通信开销开始超过了并行计算带来的收益, 即 Amdahl 定律.
- 随着矩阵规模的增加, 程序的运行时间也在增加. 矩阵乘法的计算复杂度为 $O(n^3)$, 所以当矩阵规模增加时, 所需的计算量也会显著增加.
- 程序在多进程下表现出了良好的性能提升, 但当进程数量增加到一定程度后, 通信开销可能会开始影响性能.
- 代码的扩展性受限于线程数量, 当线程数量有限时, 并行化带来的收益并不显著.

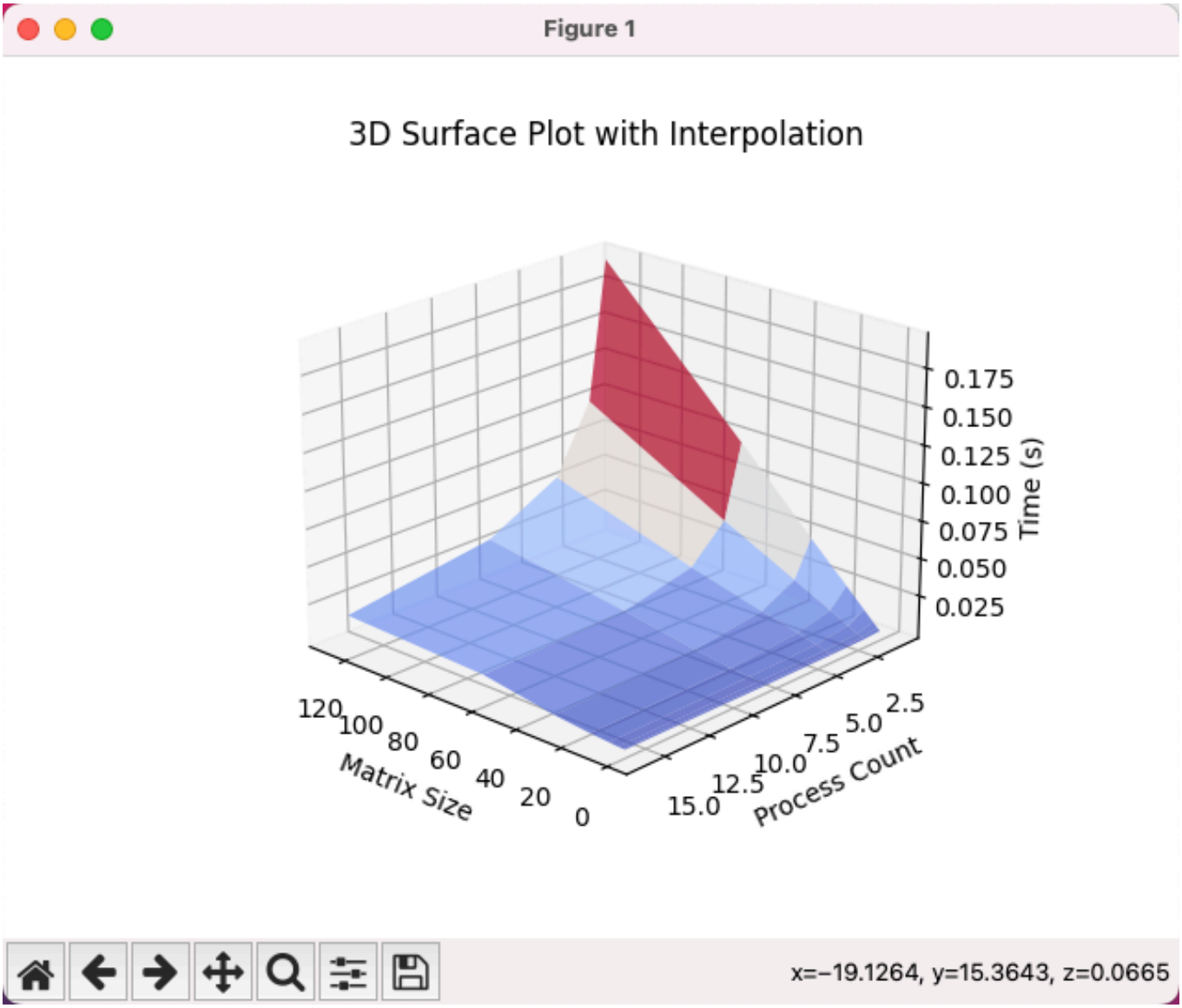
5.2 并行数组求和

以 8 核心 64M 矩阵为例

```
Mon  8 Apr - 18:12  ~/GitHub/Parallel-Programming/Task3  origin  main 4●
chef@ChefMichelin-PC  ./PthreadVecSum
64
Calculating sum for array of size 64 million
Running on 8 processes
vecAdd time: 0.015485 seconds
Verifying with Python script
The sum is correct.
```

其中 `vecSum time` 为并行矩阵乘法运行时间.

进程数量/数组规模 (s)	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
1	0.001673	0.003130	0.006023	0.012114	0.024286	0.048212	0.097572	0.193742
2	0.000942	0.001744	0.003204	0.006425	0.012462	0.024694	0.048826	0.100554
4	0.000673	0.000970	0.001919	0.003511	0.006965	0.013059	0.026101	0.056088
8	0.000492	0.000740	0.001139	0.002205	0.003717	0.006917	0.015534	0.031054
16	0.000793	0.000775	0.000954	0.001465	0.002427	0.005089	0.013129	0.017236



- 随着进程数量的增加, 程序的运行时间在大多数情况下都有所减少.
- 当进程数量增加到 16 时, 对于规模为 128M 的矩阵, 运行时间并没有显著减少. 这是因为进程间的通信开销开始超过了并行计算带来的收益, 即 Amdahl 定律.
- 随着数组规模的增加, 程序的运行时间也在增加. 数组求和的计算复杂度为 $O(n)$, 所以当数组规模增加时, 所需的计算量也会显著增加.
- 程序在多进程下表现出了良好的性能提升, 但当进程数量增加到一定程度后, 通信开销可能会开始影响性能.
- 代码的扩展性受限于线程数量, 当线程数量有限时, 并行化带来的收益并不显著.