lab3 5 6

课程：高性能计算
学期：2024秋
姓名：冯锦坤
学号：2023311D04

实验环境

内核：Linux 5.15.153.1-microsoft-standard-WSL2
发行版：Ubuntu 22.04.3 LTS
CPU：

- 型号：13th Gen Intel(R) Core(TM) i9-13900H
- 频率：2995.198 MHz
- 物理核数：10
- 内存：7 GB

gcc版本：11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)

lab3

问题

（1）：多个c代码中有相同的 `MY_MMult` 函数，怎么判断可执行文件调用的是哪个版本的 `MY_MMult` 函数？是makefile中的哪行代码决定的？

```
NEW  := MY_MMult

OBJS  := $(BUILD_DIR)/util.o $(BUILD_DIR)/REF_MMult.o $(BUILD_
all:
    make clean;
    make $(BUILD_DIR)/test_MMult.x
$(BUILD_DIR)/test_MMult.x: $(OBJS) defs.h
    $(LINKER) $(OBJS) $(LDFLAGS) -o $@
```

`NEW` 变量即为我写的 `MY_MMult` 函数，后面的代码只是用来证明：从下到上不断替换变量，就得到最原初的gcc编译指令。

（2）：性能数据 `_data/output_MMult0.m` 是怎么生成的？c代码中只是将数据输出到终端并没有写入文件。

```
    @echo "date = '`date`';" > $(DATA_DIR)/output_$(NEW).m
    @echo "version = '$(NEW)';" >> $(DATA_DIR)/output_$(NEW).
    $(BUILD_DIR)/test_MMult.x >> $(DATA_DIR)/output_$(NEW).m
```

使用了linux系统内置的重定向功能（将C程序的输出流重定向到文件里面）。

简单介绍和核心代码

这个方法利用cblas库，我只是写了个函数把 `cblas_dgemm` 函数包起来，使得与我后续其他优化函数接口一致。

```
void cblas_gemm(int m, int n, int k, double *A, int lda, doub
                double *C, int ldc) {
  double alpha = 1.0;
  double beta = 2.0;
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n
              B, ldb, beta, C, ldc);
}
```

lab5

运行截图

lab5多线程运算较大规模矩阵时的top命令输出

```
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
177744 bonlow    20   0  142220  34640   1772 R 100.0   0.4   0:04.35 test_MMult.x
```

简单介绍和核心代码

`thread_gemm` 主要功能是将矩阵分块，然后使用多线程并行计算结果。

1. 由于在分块的过程中不一定所有的线程计算的矩阵大小一致，所以定义了一个结构体来存储线程所需要的参数。

```
typedef struct {
    int id;              // 线程ID
    int num_threads;     // 总线程数
    int m;               // A 矩阵的行数
    int n;               // B 矩阵的列数
    int k;               // A 矩阵的列数，B 矩阵的行数
    double *a;          // 矩阵 A
    double *b;          // 矩阵 B
    double *c;          // 矩阵 C
    int lda;
    int ldb;
    int ldc;
} thread_arg_t;
```

2. 核心函数部分（介绍写在了注释里）

```
void *dgemm_thread(void *arg) {
    thread_arg_t *t_arg = (thread_arg_t *)arg;

        // 获取该线程所需要的参数
    int i, j, l;
    int lda = t_arg->lda;
    int ldb = t_arg->ldb;
    int ldc = t_arg->ldc;
        // 通过每个线程的t_id差异对矩阵进行分块, 确保没有重复计算和漏算
    int m_start = (t_arg->m / t_arg->num_threads) * t_arg->id
    int m_end = (t_arg->m / t_arg->num_threads) * (t_arg->id
    // 对最后一个线程进行特判，因为它的大小很可能和其他矩阵不一样
        if (t_arg->id == t_arg->num_threads - 1) {
        m_end = t_arg->m;  // 最后一个线程处理剩余的行
    }

    // 进行矩阵乘法的计算
    for (i = m_start; i < m_end; ++i) {
        for (j = 0; j < t_arg->n; ++j) {
            t_arg->C(i, j) = 0.0; // 初始化 C 矩阵的相应位置
            for (l = 0; l < t_arg->k; ++l) {
                t_arg->C(i, j) += t_arg->A(i, l) * t_arg->B(l
```

```
            }
        }
    }
    pthread_exit(NULL);
}
```

3. 用for循环创建需要的线程

```
    for (int i = 0; i < num_threads; ++i) {
        thread_args[i].id = i;
        thread_args[i].num_threads = num_threads;
        thread_args[i].m = m;
        thread_args[i].n = n;
        thread_args[i].k = k;
        thread_args[i].a = a;
        thread_args[i].b = b;
        thread_args[i].c = c;
        thread_args[i].lda = lda;
        thread_args[i].ldb = ldb;
        thread_args[i].ldc = ldc;

        pthread_create(&threads[i], NULL, dgemm_thread, (void
    }
```

3. 使用 `join` 函数等待所有线程的完成

```
    // 等待所有线程完成
    for (int i = 0; i < num_threads; ++i) {
        pthread_join(threads[i], NULL);
    }
```

lab6

CPU利用率

```
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
175468 bonlow    20   0  101436  34860   2000 R  98.0   0.4   0:02.95 test_MMult.x
```

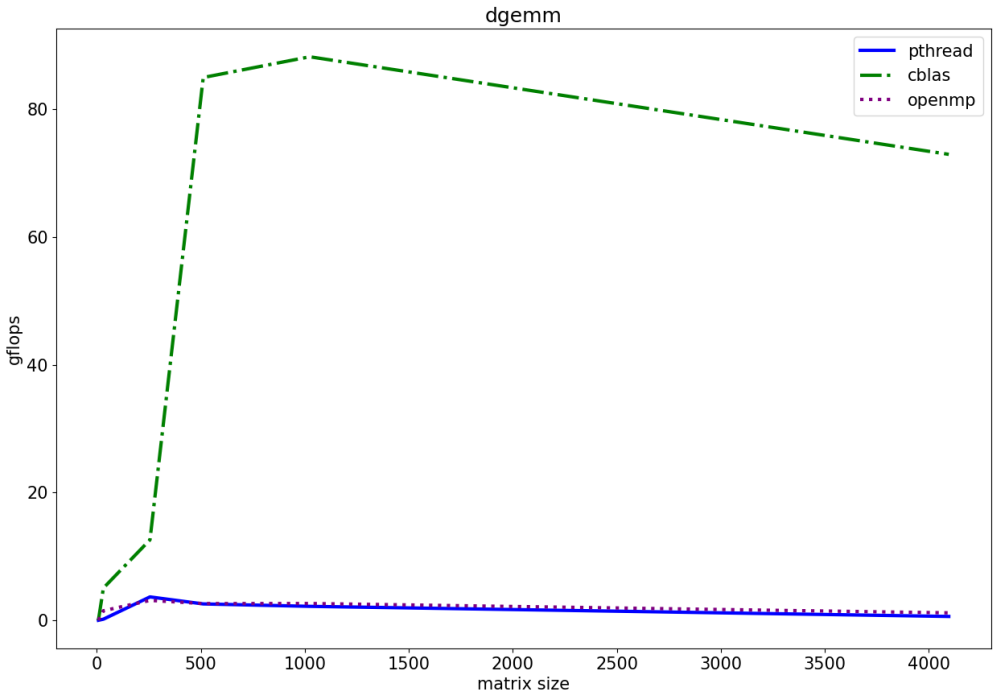
多进程证明

```
top - 13:02:48 up 12:00,  1 user,  load average: 0.00, 0.05, 0.02
Tasks:  89 total,   2 running,  87 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.0 us,  0.9 sy,  0.0 ni, 92.0 id,  0.0 wa,  0.0 hi,  7.1 si,  0.0 st
%Cpu1  :  5.3 us,  0.3 sy,  0.0 ni, 93.7 id,  0.3 wa,  0.0 hi,  0.3 si,  0.0 st
%Cpu2  :  4.6 us,  0.3 sy,  0.0 ni, 94.7 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 st
%Cpu3  :  0.3 us,  0.0 sy,  0.0 ni, 99.3 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 st
%Cpu4  :  0.7 us,  1.3 sy,  0.0 ni, 96.7 id,  1.3 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  :  5.6 us,  0.3 sy,  0.0 ni, 94.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu8  :  4.7 us,  0.0 sy,  0.0 ni, 95.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu9  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu10 :  4.3 us,  0.0 sy,  0.0 ni, 95.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu11 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu12 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu13 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu14 : 25.8 us,  0.7 sy,  0.0 ni, 73.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu15 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu16 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu17 :  4.7 us,  0.0 sy,  0.0 ni, 95.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu18 :  5.0 us,  0.0 sy,  0.0 ni, 94.7 id,  0.3 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu19 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   7804.4 total,   6507.6 free,    887.5 used,    409.3 buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used.   6617.5 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 186006 bonlow    20   0  101436  34676   1820 R  56.1   0.4   0:01.69 test_MMult.x
      1 root      20   0  167796  13136   8284 S   0.7   0.2   5:06.20 systemd
    827 root      20   0   44252  37664  10224 S   0.3   0.5   2:38.45 python3
 185871 bonlow    20   0    6348   5424   3500 S   0.3   0.1   0:00.04 bash
      2 root      20   0    2616   1444   1320 S   0.0   0.0   0:00.01 init-systemd(Ub
      7 root      20   0    2656    132    132 S   0.0   0.0   0:01.93 init
     38 root      19  -1   64284  15744  14488 S   0.0   0.2   0:00.94 systemd-journal
     60 root      20   0   23328   6764   4532 S   0.0   0.1   0:02.59 systemd-udevd
     69 root      20   0  302520   2196      0 S   0.0   0.0   0:00.01 snapfuse
     75 root      20   0  227756   2232     28 S   0.0   0.0   0:00.01 snapfuse
     79 root      20   0  152992   2192      0 S   0.0   0.0   0:00.00 snapfuse
     91 root      20   0  751104  17800    196 S   0.0   0.2   0:01.21 snapfuse
     98 root      20   0  152992   2208      4 S   0.0   0.0   0:00.00 snapfuse
    106 root      20   0  377284  11548    212 S   0.0   0.1   0:00.57 snapfuse
    111 root      20   0  153124    220      8 S   0.0   0.0   0:00.00 snapfuse
    115 root      20   0  152992   2204     12 S   0.0   0.0   0:00.00 snapfuse
    121 root      20   0  153256    180     16 S   0.0   0.0   0:00.01 snapfuse
    126 root      20   0  152992    160      0 S   0.0   0.0   0:00.00 snapfuse
    139 root      20   0  227888   2240     48 S   0.0   0.0   0:00.00 snapfuse
```

观察此时的CPU占用率，只有test_MMult运行，但是有两个CPU核心被占用，说明是多进程的。

gflops曲线图


dgemm
pthread
cblas
openmp
gflops
matrix size
0
20
40
60
80
0
500
1000
1500
2000
2500
3000
3500
4000