华中科技大学

并行编程原理与实践

专　　业：	计算机科学与技术
班　　级：	CS2005 班
学　　号：	U202090063
姓　　名：	董玲晶
指导教师：	金　海

分数	
教师签名	

2023 年 7 月 6 日

目　录

1 OpenMP 进行并行矩阵乘法

1.1 任务说明

- 你的程序应该能够接受两个矩阵作为输入，并计算它们的乘积；
- 使用 OpenMP 将矩阵乘法操作并行化，以加快计算速度；
- 考虑如何将矩阵数据进行划分和分配给不同的线程，以实现并行计算；
- 考虑如何处理并行区域的同步，以避免竞态条件和数据一致性问题；
- 考虑如何用 OpenMP 的并行循环和矩阵计算指令，进一步提高并行效率。

1.2 算法流程

首先使用 generateRandomMatrix() 函数随机生成矩阵数据集，generateRandomMatrix 函数接收两个参数，一个是 int *类型的存储生成矩阵的首地 matrix，另一个是 int 类型的矩阵大小 dim。

函数内部使用了随机设备 rd、mt19937 引擎对象 gen 和均匀分布对象 dis。在循环中，遍历矩阵的每一行和列，并使用 dis 和 gen 生成一个 1 到 100 之间的随机整数，将其赋值给矩阵中的对应元素。生成的矩阵大小为 dim × dim，代码如下。

```
void generateRandomMatrix(int *matrix, int dim) {
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_int_distribution<int> dis(1, 100);
  for (int i = 0; i < dim; ++i)
    for (int j = 0; j < dim; ++j)
      matrix[i * dim + j] = dis(gen);    }
```

代码 1.1 随机生成矩阵数据集函数

之后为了方便后续使用，调用 saveMatrixToFile()函数将指定矩阵保存至指定文件里。该函数接收三个参数，一个是 const int *类型的存储矩阵数据的指针 matrix，一个是 int 类型的矩阵大小 dim，以及 const std::string&类型的文件名 filename。该函数将矩阵的数据逐行写入到指定的文件中。如果文件成功打开，则通过双重循环遍历矩阵的每个元素，将其写入文件，并在元素之间加上空格，每行结束后写入换行符；如果文件打开失败，则输出错误消息。代码如下。

```
void saveMatrixToFile(const int *matrix, int dim, const std::string& filename) {
  std::ofstream file(filename);
  if (file.is_open()) {
    for (int i = 0; i < dim; ++i) {
      for (int j = 0; j < dim; ++j) {
        file << matrix[i * dim + j] << " ";
      }
      file << std::endl;
    }
    file.close();
  } else {
    std::cerr << "Unable to open file: " << filename << std::endl;
  }
}
```

代码 1.2 保存代码至指定文件函数

矩阵乘法选用了三种方案来对比比较，第一种是不做任何优化的基础暴力乘法，第二种是使用 OpenMP 指令进行多线程并行执行，最后一种是对矩阵进行分块处理同时也使用 OpenMP 指令并行处理。下面分别对三种方案进行详细说明。

不做任何优化处理，调用 matrix_multiply(int A[], int B[], int C[], int dim)函数，各个参数分别表示相乘的两个矩阵以及保存结果矩阵，相乘矩阵大小。该函数通过

三重循环遍历两个矩阵 A 和 B 的元素，并进行乘法运算，将结果累加得到矩阵乘积的对应元素。循环嵌套的第一层和第二层用于遍历结果矩阵 C 的每个元素，第三层循环用于遍历两个矩阵的对应行和列，并计算乘积的累加和。最后，将累加和赋值给结果矩阵 C 的对应元素。函数执行完毕后，结果矩阵 C 将包含两个矩阵乘积的结果，代码如下。

```
void matrix_multiply(int A[], int B[], int C[], int dim) {
  for (int i = 0; i < dim; i++) {
    for (int j = 0; j < dim; j++) {
      int sum = 0;
      for (int k = 0; k < dim; k++) {
        sum += A[i * dim + k] * B[k *dim + j];
      }
      C[i * dim + j] = sum;
    }
  }
}
```

代码 1.3 矩阵乘法暴力解法

使用 OpenMP 指令进行多线程并行执行，就是在暴力求解的算法上使用 #pragma omp parallel for 指令来并行化外层循环，使不同的线程可以同时计算不同的行，并行地执行矩阵乘法的计算。

矩阵分块和 OpenMP 并行，根据系统结构课程所学，矩阵分块的基本思想是将大的矩阵乘法问题划分为较小的块，通过并行计算每个块的乘法操作，以提高计算效率。这种分块的方法可以减少缓存未命中的次数，从而提高访问数据的局部性和缓存利用率。首先指定分块的大小为 32，这里其实一开始分别选用了 16、32、64，后来发现 32 性能较为好，所以最终选择了 32。在函数内部，使用两层嵌套的循环遍历结果矩阵的每个块，而对于每个块，为了确保在分块计算过程中

只处理当前块内的元素，避免越界访问，要根据起始行列索引和块的大小，计算出当前块的起始和结束索引。接着使用三层嵌套的循环遍历两个输入矩阵的相应块，并执行乘法操作。在上述分块操作之外，再加上并行化外层两个循环的操作，使用不同的线程同时处理不同的块，从而加速计算过程。代码如下。

```
Void matrix_multiply_block(int A[], int B[], int C[], int dim) {
  // 使用 OpenMP 指令并行化计算过程
  #pragma omp parallel for collapse(2)
  for (int I = 0; I < dim; I += block_size) {
for (int j = 0; j < dim; j += block_size) {
      for (int k = 0; k < dim; k += block_size) {
        // 计算当前块的起始和结束索引
        int i_end = I + block_size < dim ? I + block_size : dim;
        int j_end = j + block_size < dim ? j + block_size : dim;
        int k_end = k + block_size < dim ? k + block_size : dim;
        // 在当前块内进行矩阵乘法计算
        for (int ii = I; ii < i_end; ++ii)
          for (int jj = j; jj < j_end; ++jj)
            for (int kk = k; kk < k_end; ++kk)
              C[ii * dim + jj] += A[ii * dim + kk] * B[kk * dim + jj];
      }
}
  }
}
```

代码 1.4 矩阵乘法分块解法

1.3 算法分析

分析之前，先统一对参数进行约定，设矩阵大小为N，并行线程个数为p，原始的串行矩阵乘法算法的执行时间为T'，并行化后使用p个线程执行并行矩阵乘法算法的执行时间为T'，同时每个线程的执行时间相同。

暴力解法和单纯使用 OpenMP 指令的基础算法相同，现对此基础算法进行时空复杂度的分析。

时间复杂度，外层循环的迭代次数为 N，内层两个循环的迭代次数也都为 N，因此总的迭代次数为 N^3。在每次迭代中，内部循环进行 N 次乘法操作和 N 次累加操作，故内层循环的总操作次数为 $2*N$。由上可得，整个算法的时间复杂度为 $\boldsymbol{O(N^3)}$。

空间复杂度，使用了三个数组 A、B 和 C 来存储矩阵数据，它们的空间复杂度都是 $\boldsymbol{O(N^2)}$，所以总的空间复杂度为 $\boldsymbol{O(N^2)}$。

现对暴力解法使用并行优化后的算法进行加速比分析。因为每个线程的执行时间相同，每个线程执行的迭代次数为 N/p；由于每个迭代的计算复杂度为 $\boldsymbol{O(N)}$，因此每个线程的执行时间为 $\boldsymbol{O(N^2/P)}$。并行化后的总执行时间可以近似表示为：

$$\boldsymbol{T' = O(N^2/P)}$$

故加速比为：

$$\boldsymbol{S = \frac{T_0}{T'} \leq \frac{T_0}{O\left(\frac{N^2}{P}\right)} = \frac{P * T_0}{O(N^2)} = \frac{p * O(N^2)}{O(N^2)} = p}$$

即并行化算法的加速比上界为p，即最多可以加速p倍。

1.4 实验方案与结果分析

本实验在 Educoder 平台完成暴力算法通关后，其它操作均在本地环境完成。

根据章节 1.2 中描述的数据集生成函数随机生成不同大小（矩阵边长）的数据集（数据集见实验压缩包附录），这里分别生成了 1k~4k，步长为 0.5k 的数据集。对于每个数据集，分别使用上述三个算法进行测试，在调用前后获取时间戳，相减计算时间差。为了放大观察，对于每个算法，将矩阵乘法的调用进行十次。实验后得到如下结果。

表 1.1 不同算法下十次矩阵乘法耗时（单位：s）

矩阵大小	Baseline	OpenMP	分块
1k×1k	46.53	16.529	12.62
1.5k×1.5k	230.483	86.225	48.274
2k×2k	779.347	190.439	124.085
2.5k×2.5k	1414.18	359.067	223.803
3k×3k	2932.61	857.639	372.692
3.5k×3.5k	4294.26	1202.43	625.25
4k×4k	5558.54	1826.19	919.865

实验结果图如下所示。可以看到在没有任何优化技术下，由于串行的执行，实验耗时呈现指数级增长。使用 OpenMP 并行化的方法相比基准方法显著提高了性能，由于 OpenMP 利用多线程并行计算矩阵乘法，加快了计算速度在所有矩阵大小下，使得 OpenMP 版本的耗时都明显低于基准方法。

而与此同时可以观察到，与 OpenMP 相比，分块方法能够进一步减少计算时间，在性能上表现出更好的结果。这是因为分块方法将大矩阵分解为较小的块，并利用缓存的局部性原理减少了数据访问时间。随着矩阵大小的增加，分块方法的性能优势变得更加明显。

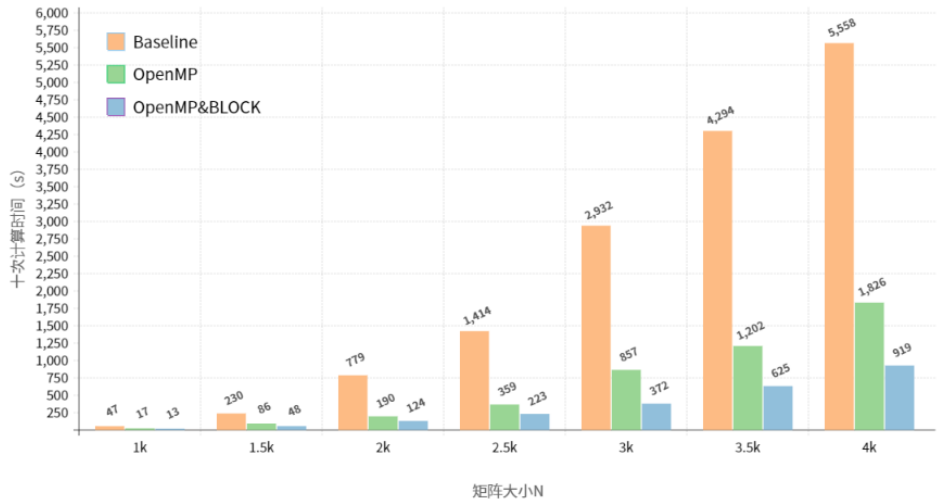

图 1.1 不同算法下十次矩阵乘法耗时直方图

2 使用 Pthreads 实现并行文本搜索

2.1 任务说明

- 你需要实现一个函数，该函数接受一个目标字符串和一个包含多个文本文件的文件夹路径；
- 程序应该并行地搜索每个文本文件，查找包含目标字符串的行，并将匹配的行打印出来；
- 每个线程应该处理一个文件，你需要合理地分配文件给不同的线程；
- 确保你的程序是线程安全的，并正确处理多个线程之间的同步问题。

2.2 算法流程

首先定义线程数据结构，包含待搜索的数据集文件名和搜索的目标字符串，代码如下。

```
typedef struct {
const char* filename;
const char* target_string;
} ThreadData;
```

代码 2.1 线程搜索数据数据结构

完成搜索单文件函数 search_file(void *arg)，该函数接收一个指向线程数据结构的指针 arg 作为参数，并将其转换为 ThreadData 类型；使用 fopen 函数以只读模式打开指定的文件 data->filename。如果打开文件失败，输出错误信息并终止当前线程。在函数内部定义一个字符数组 line，用于存储读取的每一行文本；初始化行号 line_number 为 1，用于跟踪文件中匹配目标字符串的行的行号。

进入循环，在每次迭代中读取文件的一行内容，使用 fgets 函数将其存储在字符数组 line 中。在每一行中，使用 strstr 函数搜索目标字符串

data->target_string 是否出现。如果找到目标字符串，则输出包含目标字符串的行的信息，包括文件名、行号和内容，使用 printf 函数进行输出。如此往复，读取下一行，直到到达文件末尾。

搜索完毕后，使用 pthread_exit 函数终止当前线程。代码如下。

```
void* search_file(void* arg) {
  ThreadData* data = (ThreadData*)arg;
  char line[MAX_LINE_LENGTH];
  FILE* file = fopen(data->filename, "r");
  if (file == NULL) {
    perror("Error opening file");
    pthread_exit(NULL);
  }
  int line_number = 1;
  // 执行文本搜索
  while (fgets(line, MAX_LINE_LENGTH, file) != NULL) {
    if (strstr(line, data->target_string) != NULL)
      printf("%s:%d: %s", data->filename, line_number, line);
    line_number++;
  // 文件关闭&线程退出
  fclose(file);
  pthread_exit(NULL);
}
```

代码 2.2 单个文件关键词搜索函数

完成搜索单文件函数 search_files(const char* folder_path, const char* target_string)，该函数接收接受两个参数，文件夹路径 folder_path 和目标字符串 target_string。 首先，通过调用 opendir 函数打开指定路径的文件夹，将返回的目录指针存储在 directory 变量中。如果打开文件夹失败，输出错误信息并返回。

在 while 循环判断条件中，使用 readdir 函数遍历文件夹中的每个目录项，在

每次迭代中获取下一个目录项，并将其存储在 entry 变量中。对于每个目录项，首先通过判断 entry->d_type 的值来检查其类型是否为常规文件。如果目录项是一个常规文件，则创建一个线程来执行文件的搜索任务：每个线程独立地打开文件，并逐行读取内容，在每一行中搜索目标字符串。如果找到匹配的行，则输出相应的信息。完成文件搜索后，线程退出并释放相关资源。代码如下。

```
void search_files(const char* folder_path, const char* target_string) {
  DIR* directory;
  struct dirent* entry;
  directory = opendir(folder_path);
  if (directory == NULL) {
    perror("Error opening directory");
    return;   }
  pthread_t threads[256];
  ThreadData thread_data[256];
  int num_threads = 0;
  while ((entry = readdir(directory)) != NULL) {
    // 为每个线程分配任务
    if (entry->d_type == DT_REG) {
      char file_path[MAX_LINE_LENGTH];
      snprintf(file_path, sizeof(file_path), "%s/%s", folder_path, entry->d_name);
      thread_data[num_threads].filename = strdup(file_path);
      thread_data[num_threads].target_string = target_string;
      pthread_create(&threads[num_threads],
NULL, search_file, &thread_data[num_threads]);
      num_threads++; }}
  closedir(directory);
  // 线程同步
  for (int i = 0; i < num_threads; i++) {
    pthread_join(threads[i], NULL);
    free((void*)thread_data[i].filename);}}
```

代码 2.3 多线程并行处理文件搜索函数

2.3 算法分析

分析之前，也是先统一对参数进行约定，设待搜索文件大小为 n 字节，文件的行数为 m，文件夹中的目录项个数为 k。

search_file 函数时间复杂度，由于该函数会逐行读取文件内容，并在每一行中执行字符串搜索操作。遍历每个字节读取文件的时间复杂度为 $O(n)$，在每一行中遍历每行字符进行搜索的时间复杂度近似为 $O(m)$。由上可得，总时间复杂度约为 $O(n+m)$.

search_file 函数空间复杂度，由于该函数使用固定大小的字符数组 line 来存储每行文本，空间复杂度为 $\mathrm{O(MAX\_LINE\_LENGTH)}$，即 $O(1)$。

search_files 函数时间复杂度，遍历文件夹中的每个目录项，时间复杂度为 $O(k)$，在线程同步部分等待每个线程的完成时的时间复杂度为 $O(num\_threads) \leq O(k)$，忽略创建进程的时间，总的时间复杂度为 $O(2*k)$，即 $O(k)$.

search_files 函数空间复杂度，固定大小的线程数组 threads 和线程数据结构数组 thread_data，它们的空间复杂度为 $O(1)$，而线程数据结构中的 filename 所占用的空间会在使用完毕线程结束后进行释放，因此可以忽略不计。由上可得，总的空间复杂度为 $O(1)$.

2.4 实验方案与结果分析

本实验使用的数据集为 Educoder 提供的 txt 文件数据集（包含 cn.txt 和 us.txt 两个文件），实验也是在 Educoder 完成代码编写实现通关。

输入字符串“武汉”，最终输出“/data/workspace/myshixun/texts/cn.txt:386:武汉”。表示在文件 cn.txt 的 386 行搜索到了该字符串，成功输出。

示意图如下所示。

图 2.1 第二关通关成功展示

声明：本实验所有关卡代码以及实验记录已开源至如下仓库地址，可自行验收查看，https://github.com/Elubrazione/parallel_principle_labs_hust.

3 实验小结

本次实验重新复习了计算机体系结构的矩阵乘法分块算法，同时还采用了OpenMP 并行指令来针对矩阵乘法进行了算法性能改进与测试，对比了它们在不同矩阵大小下的耗时，并对结果进行了分析。实验结果表明，通过并行化和优化技术，可以显著提高矩阵乘法算法的执行效率，尤其在处理大规模矩阵时更加明显。通过这个实验，我深入理解了并行计算的优势，并学习了如何利用 OpenMP 指令来处理并行化，提高算法的性能。同时，我还了解到了不同的算法实现方式对性能的影响，以及如何选择适当的实现方式和优化策略来满足特定的需求。

在实验二中，通过一个利用多线程并行处理文件搜索的算法，我实现了一个多线程并行文本搜索任务。在这个实验中，我重新复习了多线程编程和线程同步的概念，学会了使用线程库来创建和管理线程、实现线程间的数据共享和同步，从而达到确保线程之间的正确执行顺序和输出结果的一致性的目的。

总的来说，这两个实验使我更深入地了解了并行计算和多线程编程的概念、技术和应用。学到如何利用并行化和优化技术提高算法性能的同时，还学会了如何进行性能测试和分析，以评估和改进算法的执行效率。这些经验对于处理大规模数据和计算密集型任务具有重要意义，为未来可能的进一步探索打下了坚实的基础。