中山大学计算机院本科生实验报告

（2024 学年春季学期）

课程名称：并行程序设计	批改人:
实验：6	专业（方向）：计算机科学与技术
学号：22336226	姓名：王泓沣
Email：wanghf59@mail2.sysu.edu.cn	完成日期：2025/5/1

1 实验目的

1.1 构造基于 Pthreads 的并行 for 循环分解、分配、执行机制

模仿 OpenMP 的 omp_parallel_for 构造基于 Pthreads 的并行 for 循环分解、分配及执行机制。

问题描述：生成一个包含 `parallel_for` 函数的动态链接库（.so）文件，该函数创建多个 Pthreads 线程，并行执行 `parallel_for` 函数的参数所指定的内容。

要求：完成 `parallel_for` 函数实现并生成动态链接库文件，并以矩阵乘法为例，测试其实现的正确性及效率。

```
parallel_for(int start, int end, int inc, void *(*functor)( int,void*),
   void *arg, int num_threads)
```

Listing 1: parallel_for 函数基础定义

1.2 `parallel_for` 并行应用

使用此前构造的 parallel_for 并行结构，将 heated_plate_openmp 改造为基于 Pthreads 的并行应用。

heated plate 问题描述：规则网格上的热传导模拟，其具体过程为每次循环中通过对邻域内热量平均模拟热传导过程，即：

$$w_{i,j}^{t+1} = \frac{1}{4}(w_{i-1,j-1}^{t} + w_{i-1,j+1}^{t} + w_{i+1,j-1}^{t} + w_{i+1,j+1}^{t})$$

要求：使用此前构造的 `parallel_for` 并行结构，将 `heated_plate_openmp` 实现改造为基于 Pthreads 的并行应用。测试不同线程、调度方式下的程序并行性能，并与原始 `heated_plate_openmp` 实现对比。

2 实验过程和核心代码

2.1 构造基于 Pthreads 的并行 for 循环分解、分配、执行机制

首先在 parallel_for.h 中给出函数声明

```
#ifndef PARALLEL_FOR_H
#define PARALLEL_FOR_H

// 并行 for 的函数签名
void parallel_for(int start, int end, int inc,
                  void (*functor)(int, void*),
                  void* arg, int num_threads);

#endif
```

Listing 2: parallel_for.h

之后在 parallel_for.c 中给出具体实现:

```
typedef struct {
    int start, end, inc;
    void (*functor)(int, void*);
    void *arg;
} ThreadData;
```

Listing 3: 线程结构体

- start,end,inc 分别为循环的开始、结束及索引自增量;
- functor 为函数指针，定义每次循环所执行的内容;
- arg 为 functor 的参数指针，给出 functor 执行所需的数据。

```
static void *thread_func(void *p) {
    ThreadData *d = (ThreadData*)p;
    for (int i = d->start; i < d->end; i += d->inc) {
        d->functor(i, d->arg);
    }
    return NULL;
}
```

Listing 4: 每个线程创建后执行的函数

```
void parallel_for(int start, int end, int inc,
                  void (*functor)(int, void*),
```

```c
                    void* arg, int num_threads) {
    pthread_t *ths = malloc(num_threads * sizeof(pthread_t));
    ThreadData *td = malloc(num_threads * sizeof(ThreadData));

    int total = (end - start + inc - 1) / inc;
    int base  = total / num_threads;
    int extra = total % num_threads;
    int cur   = start;

    for (int t = 0; t < num_threads; ++t) {
        int cnt = base + (t < extra ? 1 : 0);
        td[t].start   = cur;
        td[t].end     = cur + cnt * inc;
        td[t].inc     = inc;
        td[t].functor = functor;
        td[t].arg     = arg;
        pthread_create(&ths[t], NULL, thread_func, &td[t]);
        cur = td[t].end;
    }

    for (int t = 0; t < num_threads; ++t) {
        pthread_join(ths[t], NULL);
    }
    free(ths);
    free(td);
}
```

Listing 5: parallel_for 函数实现

在终端中用编译器生成动态链接库.so 文件后，使用矩阵乘法测试程序进行测试。

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "parallel_for.h"

// functor 参数结构
typedef struct {
    int M, N, P;
    float *A, *B, *C;
} MatMulArgs;

// functor: 对单个元素 (row, col) 进行乘加
```

```
void matmul_functor(int idx, void *arg) {
    MatMulArgs *a = (MatMulArgs*)arg;
    int row = idx / a->P;
    int col = idx % a->P;
    float sum = 0;
    for (int k = 0; k < a->N; ++k) {
        sum += a->A[row * a->N + k] * a->B[k * a->P + col];
    }
    a->C[row * a->P + col] = sum;
}

int main() {
    int sizes[] = {128, 256, 512, 1024, 2048};
    int thread_counts[] = {1, 2, 4, 8, 16};

    for (int si = 0; si < sizeof(sizes)/sizeof(sizes[0]); ++si) {
        int size = sizes[si];
        for (int ti = 0; ti < sizeof(thread_counts)/sizeof(thread_counts
            [0]); ++ti) {
            int threads = thread_counts[ti];

            float *A = malloc(sizeof(float) * size * size);
            float *B = malloc(sizeof(float) * size * size);
            float *C = malloc(sizeof(float) * size * size);

            for (int i = 0; i < size * size; ++i) A[i] = (float)(rand())
                / RAND_MAX;
            for (int i = 0; i < size * size; ++i) B[i] = (float)(rand())
                / RAND_MAX;

            MatMulArgs args = {size, size, size, A, B, C};
            int total = size * size;

            struct timespec t0, t1;
            clock_gettime(CLOCK_MONOTONIC, &t0);

            parallel_for(0, total, 1, matmul_functor, &args, threads);

            clock_gettime(CLOCK_MONOTONIC, &t1);
            double elapsed = (t1.tv_sec - t0.tv_sec)
                           + (t1.tv_nsec - t0.tv_nsec) * 1e-9;
```

```
            printf("Size=%d, Threads=%d, Time=%.6f s\n", size, threads,
               elapsed);

            free(A);
            free(B);
            free(C);
        }
    }
    return 0;
}
```

Listing 6: matrix_mul_test.c 测试程序

2.2 parallel_for 并行应用

盘子参数可表示为点数 N 和加热前后所有点的温度：

```
typedef struct {
  int N;
  double *old_grid;
  double *new_grid;
} PlateArgs;
```

Listing 7: 盘子参数结构体

根据公式设计温度更新函数：

```
void update_point(int idx, void *arg) {
  PlateArgs *pa = (PlateArgs *)arg;
  int N = pa->N;
  int i = idx / N;
  int j = idx % N;
  /* skip boundary cells */
  if (i == 0 || i == N - 1 || j == 0 || j == N - 1) return;
  pa->new_grid[i * N + j] =
      0.25 * (pa->old_grid[(i - 1) * N + j] +
              pa->old_grid[(i + 1) * N + j] +
              pa->old_grid[i * N + j - 1] +
              pa->old_grid[i * N + j + 1]);
}
```

Listing 8: 温度更新函数

在主函数中，使用 parallel_for 分配各线程计算工作，线程执行函数为温度更新函数。

```c
int main ( int argc, char *argv[] )
{
  int thread_counts[] = {1, 2, 4, 8, 16};
  int num_options = sizeof(thread_counts) / sizeof(thread_counts[0]);
  int grid_sizes[] = {64, 128, 256, 512, 1024};
  int num_sizes = sizeof(grid_sizes) / sizeof(grid_sizes[0]);

  for (int s = 0; s < num_sizes; ++s) {
    int N = grid_sizes[s];

    for (int t = 0; t < num_options; ++t) {
      int num_threads = thread_counts[t];

      double *old_grid = malloc(N * N * sizeof(double));
      double *new_grid = malloc(N * N * sizeof(double));

      if (old_grid == NULL || new_grid == NULL) {
        printf("Memory allocation failed\n");
        return 1;
      }

      // Initialize grids
      memset(old_grid, 0, N * N * sizeof(double));
      memset(new_grid, 0, N * N * sizeof(double));

      // Set boundary values
      for (int i = 0; i < N; i++) {
        old_grid[i] = 100.0;
        new_grid[i] = 100.0;
        old_grid[(N-1)*N + i] = 100.0;
        new_grid[(N-1)*N + i] = 100.0;
        old_grid[i*N] = 0.0;
        new_grid[i*N] = 0.0;
        old_grid[i*N + N - 1] = 0.0;
        new_grid[i*N + N - 1] = 0.0;
      }

      // Initialize interior points to 0
      for (int i = 1; i < N - 1; i++) {
        for (int j = 1; j < N - 1; j++) {
```

```
        old_grid[i*N + j] = 0.0;
        new_grid[i*N + j] = 0.0;
      }
    }

    struct timeval start, end;
    gettimeofday(&start, NULL);

    double epsilon = 0.01;
    double diff;
    int iterations = 0;

    do {
      /* parallel stencil update */
      PlateArgs args = { N, old_grid, new_grid };
      parallel_for(0, N * N, 1, update_point, &args, num_threads);

      /* sequential reduction to obtain max-difference */
      diff = 0.0;
      for (int i = 1; i < N - 1; i++) {
        for (int j = 1; j < N - 1; j++) {
          double delta = fabs(new_grid[i * N + j] - old_grid[i * N + j
             ]);
          if (delta > diff) diff = delta;
        }
      }

      /* swap grids */
      double *temp = old_grid;
      old_grid = new_grid;
      new_grid = temp;
      iterations++;
    } while (diff > epsilon);

    gettimeofday(&end, NULL);
    double elapsed = (end.tv_sec - start.tv_sec) + (end.tv_usec - start
       .tv_usec) / 1000000.0;

    double checksum = 0.0;
    for (int i = 0; i < N * N; i++) {
      checksum += old_grid[i];
```

```
    }

    printf("GridSize=%d, Threads=%d, Time=%.6f s, Checksum=%f\n", N,
        num_threads, elapsed, checksum);

    free(old_grid);
    free(new_grid);
  }
}

return 0;
}
```

Listing 9: main 函数

3 实验结果

3.1 构造基于 Pthreads 的并行 for 循环分解、分配、执行机制

表 1: 使用基于 Pthreads 并行 for 循环机制的矩阵乘法耗时（秒）

矩阵规模	单线程	2 线程	4 线程	8 线程	16 线程
128	0.003555	0.001830	0.001017	0.000906	0.001280
256	0.029223	0.016917	0.008829	0.009271	0.007554
512	0.247293	0.147641	0.099897	0.078469	0.069139
1024	2.189065	1.160178	0.774294	0.643101	0.629591
2048	21.522023	13.913186	7.880217	5.773248	5.843289

3.2 parallel_for 并行应用

表 2: OpenMP Static 调度下的执行时间（秒）

Size	1 Th	2 Th	4 Th	8 Th	16 Th
64	0.012819	0.031753	0.041508	0.101474	0.188289
128	0.140609	0.162372	0.151400	0.322770	0.552198
256	1.374303	0.972642	0.660889	1.041841	1.561530
512	9.304864	5.308769	3.090632	3.678414	4.516667
1024	39.574823	20.847362	11.924667	12.572939	12.598300

表 3: OpenMP Dynamic 调度下的执行时间（秒）

Size	1 Th	2 Th	4 Th	8 Th	16 Th
64	0.013630	0.031561	0.040159	0.096555	0.189232
128	0.146327	0.160892	0.148317	0.314957	0.586071
256	1.430952	0.975445	0.677916	1.038554	1.585976
512	9.324221	5.304894	3.107984	3.694447	4.541478
1024	39.632346	20.859141	12.301450	12.998093	12.996476

表 4: OpenMP Guided 调度下的执行时间（秒）

Size	1 Th	2 Th	4 Th	8 Th	16 Th
64	0.016051	0.033071	0.038219	0.090641	0.185977
128	0.151275	0.160899	0.148453	0.312236	0.536992
256	1.425200	1.009398	0.664623	1.045498	1.567207
512	9.366057	5.292338	3.154919	3.778247	4.676591
1024	39.642213	20.914889	12.551449	13.152857	13.051860

表 5: Pthreads 参考实现的运行时间与校验和

GridSize	1 Th	2 Th	4 Th	8 Th	16 Th	Checksum
64	0.017024	0.050944	0.058606	0.090631	0.165899	191782.076406
128	0.088303	0.123934	0.135628	0.175214	0.259005	565924.036589
256	0.341600	0.350840	0.361443	0.359527	0.425772	1286824.076029
512	1.357413	1.244185	1.322787	1.192774	1.231026	2733901.276734
1024	5.424402	5.251755	5.084450	4.585383	4.502328	5627598.124312

4 实验感想

在设计 `parallel_for` 函数时，每个线程所需执行的函数 functor 是重点。将具体任务抽象成一个 functor 是提升程序可迁移性的关键。