中山大学计算机院本科生实验报告

(2024学年秋季学期)

课程名称：高性能计算程序设计

实验	并行计算性能优化与分析	专业(方向)	信息与计算科学
学号	**22336313**	**姓名**	**郑鸿鑫**
Email	zhenghx57@mail2.sysu.edu.cn	**完成日期**	**2024/11/25**

1. 实验目的

本次实验的核心内容是分别用不同的实现方式来实现并行化计算热传导模拟中的迭代计算部分，目的在于掌握Phreads和Openmp还有MPI这几种不同实现方式对同一个程序如何做到分解任务，分配资源和进程/线程间的通信，从而做到并行化执行程序，并对不同规模和不同进程/线程数的执行时间和内存消耗做对比，评估它们的性能。目的在于理解并掌握不同的并行实现方式的实现并理解它们之间的区别，提高使用并行计算来解决问题的能力。

2. 实验过程和核心代码

> **子任务1** 通过实验3构造的基于Pthreads的parallel_for函数替换heated_plate_openmp应用中的某些计算量较大的“for循环” ,实现for循环分解、分配和线程并行执行。

复用上次实验的parallel.c，并且编写新的文件heated_plate_parallel.c，将openmp版本中的迭代计算部分的循环用上次实验的parallel_for函数来实现并行：
代码如下：

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "parallel.h" // 引入自定义的并行库
#include <pthread.h>
#include <time.h>
#define M 500 // 矩阵的行数
#define N 500 // 矩阵的列数
#define max(a,b) ((a) > (b) ? (a) : (b)) // 定义求最大值的宏
// 自定义结构体，用于传递参数到线程函数
typedef struct {
    double(*u)[N]; // 原始矩阵指针
    double(*w)[N]; // 更新后的矩阵指针
    double diff;   // 当前的最大差值
} Args;
// 获取当前时间的函数（高精度）
double get_time() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts); // 使用高精度时钟获取时间
    return ts.tv_sec + ts.tv_nsec * 1e-9; // 返回秒数（包括纳秒部分）
}
// 全局变量
int num_threads = 4; // 线程数
double u[M][N];      // 原始矩阵
double w[M][N];      // 更新后的矩阵
double diff;         // 当前最大差值
double epsilon = 0.001; // 收敛条件
double mean = 0.0;   // 边界值平均值
int iterations = 0;  // 迭代次数
int iterations_print = 1; // 控制输出的迭代次数
pthread_mutex_t diff_mutex = PTHREAD_MUTEX_INITIALIZER; // 互斥锁，用于保护 diff
// 线程函数，用于计算每一行的更新和差值
void* interation_operation(void* args, int i) {
    Args* a = (Args*)args;
    double(*u)[N] = a->u;
    double(*w)[N] = a->w;
    double local_diff = 0.0; // 每个线程的局部最大差值
    // 更新矩阵的第 i 行（不包括边界）
    for (int j = 1; j < N - 1; j++) {
```

```
        w[i][j] = (u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1]) / 4.0; // 计算新值
        local_diff = max(local_diff, fabs(w[i][j] - u[i][j])); // 计算差值的绝对值
    }
    // 使用互斥锁保护全局变量 diff 的更新
    pthread_mutex_lock(&diff_mutex);
    a->diff = max(a->diff, local_diff); // 更新全局最大差值
    pthread_mutex_unlock(&diff_mutex);
    return NULL;
}
int main(int argc, char* argv[]) {
    int i, j;
    // 输出程序信息
    printf("\nHEATED_PLATE_PARALLEL_FOR\n");
    printf("  C/PARALLEL_FOR version\n");
    printf("  A program to solve for the steady state temperature distribution\n");
    printf("  over a rectangular plate.\n");
    printf("\n");
    printf("  Spatial grid of %d by %d points.\n", M, N);
    printf("  The iteration will be repeated until the change is <= %e\n", epsilon);
    printf("  Number of processors available = %d\n", num_threads);
    printf("  Number of threads = %d\n", num_threads);
    // 初始化边界值
    for (i = 1; i < M - 1; i++) {
        w[i][0] = 100.0;   // 左边界
        w[i][N-1] = 100.0; // 右边界
    }
    for (j = 0; j < N; j++) {
        w[M-1][j] = 100.0; // 下边界
        w[0][j] = 0.0;     // 上边界
    }
    // 计算边界值的平均值
    for (i = 1; i < M - 1; i++) {
        mean += w[i][0] + w[i][N-1];
    }
    for (j = 0; j < N; j++) {
        mean += w[M-1][j] + w[0][j];
    }
    mean /= (2 * M + 2 * N - 4); // 平均值
    printf("\n  MEAN = %f\n", mean);
```

```
    // 初始化内部值为边界值的平均值
    for (i = 1; i < M - 1; i++) {
        for (j = 1; j < N - 1; j++) {
            w[i][j] = mean;
        }
    }
    // 输出迭代头信息
    printf("\n Iteration  Change\n\n");
    diff = epsilon; // 初始化最大差值
    double start_time = get_time(); // 开始计时
    // 主迭代循环，直到差值小于等于 epsilon
    while (epsilon <= diff) {
        // 将 w 的值复制到 u 中
        for (i = 0; i < M; i++) {
            for (j = 0; j < N; j++) {
                u[i][j] = w[i][j];
            }
        }
        // 创建参数结构体，并初始化 diff 为 0
        Args args = { .u = u, .w = w, .diff = 0.0 };
        parallel_for(1, M - 1, 1, interation_operation, &args, num_threads); // 并行执行每一行的更新
        // 更新全局 diff
        diff = args.diff;
        iterations++; // 增加迭代次数
        if (iterations == iterations_print) {
            // 输出当前迭代的信息
            printf("  %8d  %f\n", iterations, diff);
            iterations_print *= 2; // 下一次输出间隔加倍
        }
    }
    double end_time = get_time(); // 结束计时
    // 输出最终结果
    printf("\n  %8d  %f\n", iterations, diff);
    printf("\n  Error tolerance achieved.\n");
    printf("  Wallclock time = %f\n", end_time - start_time);
    printf("\nHEATED_PLATE_PARALLEL_FOR:\n");
    printf("  Normal end of execution.\n");
    return 0;
}
```

注： 由于需要用到矩阵的行号才能对具体的每一行做迭代更新，所以每一次执行iterations_operation函数都需要多接收一个int型参数为对应处理的行号。

> **子任务2** （二选一）
>
> 1. 将fft_serial应用改造成基于MPI的进程并行应用（为了适合MPI的消息机制，可能需要对fft_serial的代码实现做一定调整）。Bonus:使用MPI_Pack/MPI_Unpack，或MPI_Type_create_struct实现数据重组后的消息传递。
> 2. 将heated_plate_openmp应用改造成基于MPI的进程并行应用。

这个子任务我选择的是2，将上述openmp迭代的版本通过MPI的方式来实现并行化
编写代码如下：

```c
# include <stdlib.h>
# include <string.h>
# include <stdio.h>
# include <math.h>
# include <mpi.h>
// 定义矩阵的尺寸
# define M 500
# define N 500
// # define M 16
// # define N 16
// 定义最大值宏
#define max(a,b) ((a) > (b) ? (a) : (b))
// 定义全局变量
double epsilon = 0.001;     // 收敛条件，温差小于该值时停止迭代
double diff = 0.001;        // 当前温差值
int i, j;                  // 用于遍历的循环变量
int iterations = 0;         // 当前迭代次数
int iterations_print = 1;   // 控制打印频率的变量
double mean = 0;            // 初始温度均值
double my_diff;             // 每个进程的局部温差
double wtime;               // 程序运行时间
double u[M][N];             // 存储旧的温度值
double w[M][N];             // 存储新的温度值
int main ( int argc, char *argv[] ){
    int rank, comm_sz;
    // 初始化 MPI 环境
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz); // 获取进程总数
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);    // 获取当前进程编号
    if(rank == 0){
        // 主进程打印初始化信息
        printf ( "\n" );
        printf ( "HEATED_PLATE_MPI\n" );
        printf ( "  C/MPI version\n" );
        printf ( "  A program to solve for the steady state temperature distribution\n" );
        printf ( "  over a rectangular plate.\n" );
        printf ( "\n" );
        printf ( "  Spatial grid of %d by %d points.\n", M, N );
        printf ( "  The iteration will be repeated until the change is <= %e\n", epsilon );
```

```
    printf ( "  Number of processors available = %d\n", comm_sz );
    printf ( "  Number of threads =              %d\n", comm_sz );
    // 初始化矩阵边界条件
    for (i = 1; i < M - 1; i++) {
        w[i][0] = 100.0;   // 左边界
        w[i][N-1] = 100.0; // 右边界
    }
    for (j = 0; j < N; j++) {
        w[M-1][j] = 100.0; // 下边界
        w[0][j] = 0.0;     // 上边界
    }
    // 计算初始温度均值
    for (i = 1; i < M - 1; i++) {
        mean += w[i][0] + w[i][N-1];
    }
    for (j = 0; j < N; j++) {
        mean += w[M-1][j] + w[0][j];
    }
    mean /= (2 * M + 2 * N - 4); // 均值计算公式
    printf("\n  MEAN = %f\n", mean);
    // 初始化内部网格点的温度为均值
    for (i = 1; i < M - 1; i++) {
        for (j = 1; j < N - 1; j++) {
            w[i][j] = mean;
        }
    }
    printf("\n Iteration  Change\n\n");
}
// 计算每个进程负责的行范围
int range = M / comm_sz;  // 每个进程负责的基本行数
int start = rank * range; // 当前进程起始行
int end = (rank != comm_sz-1) ? (rank + 1) * range - 1 : M - 1; // 当前进程结束行
int row_nums = end - start + 1; // 当前进程的行数
// 动态分配局部网格（比实际行数多两行，用于存储邻接行数据）
double* local_u = (double*)malloc((row_nums + 2) * N * sizeof(double)); // 局部旧解
double* local_w = (double*)malloc((row_nums + 2) * N * sizeof(double)); // 局部新解
memset(local_u, 0, (row_nums + 2) * N * sizeof(double)); // 初始化为 0
memset(local_w, 0, (row_nums + 2) * N * sizeof(double)); // 初始化为 0
```

```
// 将全局网格分发到各个进程的局部网格
MPI_Scatter(
    w,                  // 发送缓冲区
    row_nums * N,       // 每个进程接收的数据大小
    MPI_DOUBLE,         // 数据类型
    &local_w[N],        // 接收缓冲区（跳过第 1 行缓冲区）
    row_nums * N,       // 接收数据大小
    MPI_DOUBLE,         // 数据类型
    0,                  // 根进程
    MPI_COMM_WORLD      // 通信域
);
MPI_Scatter(
    u,                  // 同上，分发旧解
    row_nums * N,
    MPI_DOUBLE,
    &local_u[N],
    row_nums * N,
    MPI_DOUBLE,
    0,
    MPI_COMM_WORLD
);

// 开始计时
double start_time = MPI_Wtime();

// 开始迭代
while(diff >= epsilon)
{
    // 将当前温度分布保存到局部旧解中
    memcpy(&local_u[N], &local_w[N], row_nums * N * sizeof(double));

    // 通过通信更新边界邻接行数据
    if (rank > 0) { // 如果不是第一个进程
        MPI_Sendrecv(&local_w[N], N, MPI_DOUBLE, rank - 1, 0,    // 发送第一行到上一进程
                     &local_w[0], N, MPI_DOUBLE, rank - 1, 0,    // 接收上一进程的最后一行
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    if (rank < comm_sz - 1) { // 如果不是最后一个进程
        MPI_Sendrecv(&local_w[N * row_nums], N, MPI_DOUBLE, rank + 1, 0, // 发送最后一行到下一进
```

```
                    &local_w[N * (row_nums + 1)], N, MPI_DOUBLE, rank + 1, 0, // 接收下一进程的
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    // 更新网格点温度并计算局部最大变化量
    double local_diff = 0.0;
    for(int i = 1; i < row_nums + 1; i++){
        for(int j = 1; j < N - 1; j++){
            // 跳过边界点
            if(rank == 0 && i == 1) continue;
            if(rank == comm_sz-1 && i == row_nums) continue;
            local_w[i * N + j] = (local_u[(i - 1) * N + j] + local_u[(i + 1) * N + j] +
                                  local_u[i * N + j - 1] + local_u[i * N + j + 1]) / 4.0;
            local_diff = max(local_diff, fabs(local_w[i * N + j] - local_u[i * N + j]));
        }
    }
    // 汇总所有进程的最大温差到全局
    MPI_Allreduce(&local_diff, &diff, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
    // 增加迭代次数并打印中间结果（仅主进程打印）
    iterations++;
    if(rank == 0 && iterations == iterations_print){
        printf ( "  %8d  %f\n", iterations, diff );
        iterations_print = 2 * iterations_print;
    }
}
// 记录结束时间并计算总耗时
double end_time = MPI_Wtime();
double local_time = end_time - start_time;
MPI_Allreduce(&local_time, &wtime, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
// 打印最终结果（仅主进程打印）
if(rank == 0){
    printf ( "\n" );
    printf ( "  %8d  %f\n", iterations, diff );
    printf ( "\n" );
    printf ( "  Error tolerance achieved.\n" );
    printf ( "  Wallclock time = %f\n", wtime );
    printf ( "\n" );
    printf ( "HEATED_PLATE_MPI:\n" );
    printf ( "  Normal end of execution.\n" );
}
```

```
    // 释放动态分配的内存
    free(local_u);
    free(local_w);
    // 结束 MPI 环境
    MPI_Finalize();
    return 0;
}
```

说明： 由于 MPI 是进程级的并行，各进程之间不共享内存，因此在执行计算时，每个进程只负责自身被分配的矩阵行的数据处理。然而，在进行每次迭代计算时，矩阵的某个网格点需要利用其上下左右四个相邻网格点的值计算新的温度值。对于被分配到某进程的局部矩阵部分，其中位于上下边界的行需要相邻进程提供对应的行数据，作为更新计算的边界条件。具体情况如下：

1. 中间进程（既非首进程，也非尾进程）需要从其上一个进程接收一行（上边界行）数据，从其下一个进程接收一行（下边界行）数据。
2. 首进程仅需从下一个进程接收一行（下边界行）数据。
3. 尾进程仅需从上一个进程接收一行（上边界行）数据。

为实现这一逻辑，在给每个进程分配局部矩阵时，除了分配实际负责的行数外，额外增加了两行缓冲区（上下各一行），用于存储从相邻进程接收到的边界行数据。这些额外的行分别位于局部矩阵的顶部和底部，分配的局部矩阵结构如下：

1. 第一行：用于存储从上一进程接收的边界行数据（仅非首进程需要用到）。
2. 中间行：存储本进程实际负责的网格点数据。
3. 最后一行：用于存储从下一进程接收的边界行数据（仅非尾进程需要用到）

还有一点是通信本来是采用MPI_Send和MPI_Recv实现的，但是后续发现会产生死锁（假设两个相邻进程 A 和 B 都需要交换边界行数据。如果进程 A 调用 MPI_Send 向 B 发送数据，而进程 B 同时调用 MPI_Send 向 A 发送数据，双方都在等待对方接收，造成死锁。）导致进程堵塞，所以改成通过 MPI_Sendrecv 实现，既能够发送当前进程的边界行数据给相邻进程，又能够接收相邻进程的边界行数据，从而更新缓冲区。

> **子任务3** 性能分析任务：对任务1实现的并行化应用在不同规模下的性能进行分析，即分析：
> 1）不同规模下的并行化应用的执行时间对比；
> 2）不同规模下的并行化应用的内存消耗对比。

> 本题中，“规模”定义为“问题规模”和“并行规模”；“性能”定义为“执行时间”和“内存消耗”。例如，问题规模N或者M，值为2，4，6，8，16，32，64，128，……， 2097152；并行规模，值为1，2，4，8进程/线程。

详见实验结果分析。

3. 实验结果

首先我们先给出原始的使用Openmp版本并行的结果，以方便下面对照实验结果：

```
n@XiaoxinPro:~/Lab5$ ./Openmp

HEATED_PLATE_OPENMP
  C/OpenMP version
  A program to solve for the steady state temperature distribution
  over a rectangular plate.

  Spatial grid of 500 by 500 points.
  The iteration will be repeated until the change is <= 1.000000e-03
  Number of processors available = 8
  Number of threads =              8

  MEAN = 74.949900

 Iteration  Change

         1  18.737475
         2  9.368737
         4  4.098823
         8  2.289577
        16  1.136604
        32  0.568201
        64  0.282805
       128  0.141777
       256  0.070808
       512  0.035427
      1024  0.017707
      2048  0.008856
      4096  0.004428
      8192  0.002210
     16384  0.001043

     16955  0.001000

  Error tolerance achieved.
  Wallclock time = 1.899191

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

子任务1 实验结果

先编译parallel.c为lparallel.so动态库文件：

```
gcc -shared -fPIC -o libparallel.so parallel.c -lpthread
```

然后来编译热传导迭代的代码，指定动态库路径链接上面编译好的动态库

```
gcc -O3 -o Parallel heated_plate_parallel.c -L. -lparallel -lpthread
```

最后指定链接器在当前目录查找所需的动态库,运行程序

```
LD_LIBRSRY_PATH=. ./Parallel
```

得到结果如下所示（以8线程的parallel版本并行为例，完整结果见Result文件夹）：

```
n@XiaoxinPro:~/Lab5$ gcc -fPIC -shared -o libparallel.so parallel.c -lpthread
n@XiaoxinPro:~/Lab5$ gcc -O3 -o Test2 heated_plate_parallel.c -L. -lparallel -lpthread
n@XiaoxinPro:~/Lab5$ LD_LIBRARY_PATH=. ./Test2

HEATED_PLATE_PARALLEL_FOR
  C/PARALLEL_FOR version
  A program to solve for the steady state temperature distribution
  over a rectangular plate.

  Spatial grid of 500 by 500 points.
  The iteration will be repeated until the change is <= 1.000000e-03
  Number of processors available = 8
  Number of threads = 8

  MEAN = 74.949900

 Iteration  Change

         1  18.737475
         2  9.368737
         4  4.098823
         8  2.289577
        16  1.136604
        32  0.568201
        64  0.282805
       128  0.141777
       256  0.070808
       512  0.035427
      1024  0.017707
      2048  0.008856
      4096  0.004428
      8192  0.002210
     16384  0.001043

     16955  0.001000

  Error tolerance achieved.
  Wallclock time = 9.354144

HEATED_PLATE_PARALLEL_FOR:
  Normal end of execution.
```

子任务2 实验结果

先编译编写好的MPI版本代码：

```
mpicc -O3 -o Mpi heated_plate_MPI.c
```

运行程序并用-np参数来指定运行的线程数（并且MPI版本的实现过程中默认进程数可以整除矩阵规模，所以我们只能选择1，2，4进程）

```
mpirun -np 4 Mpi
```

得到结果如下所示（以4进程的MPI版本并行为例，完整实验结果见Result文件夹）：

```
n@XiaoxinPro:~/Lab5$ mpirun -np 4 ./Mpi

HEATED_PLATE_MPI
  C/MPI version
  A program to solve for the steady state temperature distribution
  over a rectangular plate.

  Spatial grid of 500 by 500 points.
  The iteration will be repeated until the change is <= 1.000000e-03
  Number of processors available = 4
  Number of threads =              4

  MEAN = 74.949900

 Iteration  Change

         1  18.737475
         2  9.368737
         4  4.098823
         8  2.289577
        16  1.136604
        32  0.568201
        64  0.282805
       128  0.141777
       256  0.070808
       512  0.035427
      1024  0.017707
      2048  0.008856
      4096  0.004428
      8192  0.002210
     16384  0.001043

     16955  0.001000

  Error tolerance achieved.
  Wallclock time = 3.252596

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

> **子任务3 实验结果**

由于执行内存消耗命令会影响程序发挥真实性能，所以我们先单独测量矩阵的规模从128到4096，线程数从1到8的执行时间，再单独每次测量内存消耗。先将子任务1代码中多余的打印信息注释，只保留运行时间和线程数，矩阵规模等信息，编译运行命令同子任务1 ，运行结果如下（以512x512的矩阵为例，完整结果见Result文件夹）：

```
n@XiaoxinPro:~/Lab5$ gcc -O3 -o Parallel heated_plate_parallel.c -L. -lpthread -lparallel
n@XiaoxinPro:~/Lab5$ LD_LIBRARY_PATH=. ./Parallel
  Spatial grid of 512 by 512 points.
  Number of threads = 1
  Wallclock time = 85.113743
n@XiaoxinPro:~/Lab5$ gcc -O3 -o Parallel heated_plate_parallel.c -L. -lpthread -lparallel
n@XiaoxinPro:~/Lab5$ LD_LIBRARY_PATH=. ./Parallel
  Spatial grid of 512 by 512 points.
  Number of threads = 2
  Wallclock time = 43.460574
n@XiaoxinPro:~/Lab5$ gcc -O3 -o Parallel heated_plate_parallel.c -L. -lpthread -lparallel
n@XiaoxinPro:~/Lab5$ LD_LIBRARY_PATH=. ./Parallel
  Spatial grid of 512 by 512 points.
  Number of threads = 4
  Wallclock time = 28.842989
n@XiaoxinPro:~/Lab5$ gcc -O3 -o Parallel heated_plate_parallel.c -L. -lpthread -lparallel
n@XiaoxinPro:~/Lab5$ LD_LIBRARY_PATH=. ./Parallel
  Spatial grid of 512 by 512 points.
  Number of threads = 8
  Wallclock time = 10.237642
```

各矩阵规模和线程数的执行时间如下表所示：

规模 \ 线程数	1	2	4	8
128x128	0.289491	0.345509	0.529690	1.250473
256x256	1.489464	1.426044	1.638904	3.371163
512x512	85.113743	43.460574	28.842989	10.237642
1024x1024	76.178907	53.780440	55.463984	58.327927
2048x2048	263.778766	205.413495	206.511200	196.818895
4096x4096	938.439153	734.467699	762.179758	739.972064

分析：

在子任务3的性能分析中，我们观察到随着问题规模的增加，执行时间总体呈上升趋势，这是由于更大的数据集需要更多的计算资源。然而，增加线程数并不总是导致性能提升，特别是在小规模问题上，线程创建和同步的开销可能会抵消并行计算的优势，如128x128的规模，线程数越多反而运行时间越长。对于更大规模的问题，增加线程数最初可以显著减少执行时间，但随着线程数的继续增加，性能提升的幅度逐渐减小，这可能是由于线程间通信开销的增加和负载不均衡问题。特别是在1024x1024和4096x4096的规模下，4线程的执行时间异常高于2线程的情况，这表明在这些特定配置下，可能存在优化空间，如改进负载分配和减少通信开销

然后再对程序执行的内存消耗进行测量：

--tool=massif指定 Valgrind 使用 massif 工具，专门用于内存使用分析，记录内存分配和使用的

峰值;--time-unit=B指定时间单位为 B（已分配的字节数），表示随着程序分配内存量的变化来记录数据点，而不是使用时间或指令数。

```
gcc -O3 -o Parallel heated_plate_parallel.c -L. -lpthread -lparallel
LD_LIBRARY_PATH=.
valgrind --tool=massif --time-unit=B --stacks=yes ./Parallel
```

再使用ms_print命令打印内存消耗信息：

```
ms_print massif.out.18052
```

结果如下所示（以512x512为例展示，完整结果见Result文件夹）：

```
n@XiaoxinPro:~/Lab5$ gcc -O3 -o Parallel heated_plate_parallel.c -L. -lpthread -lparallel
n@XiaoxinPro:~/Lab5$ LD_LIBRARY_PATH=.
n@XiaoxinPro:~/Lab5$ valgrind --tool=massif --time-unit=B --stacks=yes ./Parallel
==1461== Massif, a heap profiler
==1461== Copyright (C) 2003-2017, and GNU GPL'd, by Nicholas Nethercote
==1461== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==1461== Command: ./Parallel
==1461==
==1461== error calling PR_SET_PTRACER, vgdb might block
  Spatial grid of 512 by 512 points.
  Number of threads = 8
  Wallclock time = 167.439139
==1461==
n@XiaoxinPro:~/Lab5$ ls massif.out.*
massif.out.1461  massif.out.17031  massif.out.18052
n@XiaoxinPro:~/Lab5$ ms_print massif.out.1461
--------------------------------------------------------------------------------
Command:            ./Parallel
Massif arguments:   --time-unit=B --stacks=yes
ms_print arguments: massif.out.1461
--------------------------------------------------------------------------------


    MB
20.89^                                                                       #
     |                                                                     @@@#
     |                                                                   @@@@@@#
     |                                                                 ::@@@@@@@@#
     |                                                               ::::::@@@@@@@@#
     |                                                             @@::: :::@@@@@@@@#
     |                                                           @@@@@::: :::@@@@@@@@#
     |                                                         @@@@@@@@::: :::@@@@@@@@#
     |                                                    @@@@@@ @@@@@@::: :::@@@@@@@@#
     |                                                 @@@:@@ @ @ @@@@@@::: :::@@@@@@@@#
     |                                              @@@@ @:@@ @ @ @@@@@@::: :::@@@@@@@@#
     |                                           @@@@@ @ @:@@ @ @ @@@@@@::: :::@@@@@@@@#
     |                                        @@@@@@ @@ @ @:@@ @ @ @@@@@@::: :::@@@@@@@@#
     |                                  @:@@@ @@@ @@ @ @:@@ @ @ @@@@@@::: :::@@@@@@@@#
     |                               @@@@@:@@@ @@@ @@ @ @:@@ @ @ @@@@@@::: :::@@@@@@@@#
     |                         ::@@@ @ @:@@@ @@@ @@ @ @:@@ @ @ @@@@@@::: :::@@@@@@@@#
     |                    @@@: @ @ @ @:@@@ @@@ @@ @ @:@@ @ @ @@@@@@::: :::@@@@@@@@#
     |               @@@:@ @: @ @ @ @:@@@ @@@ @@ @ @:@@ @ @ @@@@@@::: :::@@@@@@@@#
     |          @::@@ :@ @: @ @ @ @:@@@ @@@ @@ @ @:@@ @ @ @@@@@@::: :::@@@@@@@@#
     |    @@@@@::@@ :@ @: @ @ @ @:@@@ @@@ @@ @ @:@@ @ @ @@@@@@::: :::@@@@@@@@#
   0 +------------------------------------------------------------------------>GB
     0                                                                   3.554
```

分析：

1. 纵轴表示内存使用量，单位为 MB。图中最高点标记为 20.89 MB，说明程序在运行期间的内存使用峰值接近 21 MB。
2. 横轴显示程序的内存使用随着运行时间或分配量的变化趋势。在这张图中，横轴的最大值为 3.554 GB，这不是直接的时间，而是以分配字节数（--time-unit=B）为单位。也就是说，随着分配的内存量增加，记录了内存使用情况的变化。
3. 图中字符（@, #, : 等）
   @ 表示堆内存的分配。

: 和 # 分别代表栈内存或其他种类的内存分配。
图中 @ 和 : 的比例较高，说明程序大部分内存使用来自于堆分配和一些栈操作。

> **程序运行的内存特点**
> 堆内存增长
> 堆内存的主要使用部分（@）逐渐增加，符合并行程序分配大块数据结构（如矩阵行或任务块）的特点。
> 栈内存的使用
> 栈内存的增长较少，可能是因为每个线程的函数调用深度有限，或者栈上的数据结构较简单（例如循环变量和临时数据）。
> 内存稳定
> 内存使用峰值后未下降，表明程序可能在运行结束前没有释放内存（如一些大数组被持续使用）。

4. 实验感想

在整个高性能计算程序设计的实验过程中，我获得了宝贵的实践经验和深刻的认识，特别是在并行计算的实现、优化和性能分析方面。

在子任务1中，我通过将OpenMP应用中的for循环转换为基于Pthreads的并行实现，深入理解了线程的创建、管理和同步机制。这个过程中，我学习了如何有效地分解任务，并将它们分配给多个线程以实现并行执行。我意识到，尽管并行化可以显著提高计算效率，但线程管理的复杂性和同步开销也需要仔细考量。

子任务2要求我将热传导方程的迭代过程使用MPI进行并行化。这一挑战促使我深入研究了MPI的通信机制和数据分发策略。我学会了如何设计高效的通信模式，以及如何通过同步机制确保数据的一致性和正确性。在这个过程中，我深刻体会到了进程间通信在并行计算中的核心作用，以及它对程序性能的影响。

子任务3的性能分析任务是对整个实验的总结和检验。通过对比不同规模下并行化应用的执行时间和内存消耗，我学会了如何使用Valgrind的Massif工具来监测程序的内存使用情况。性能分析的结果帮助我理解了并行程序的性能特点，以及如何通过调整并行规模和问题规模来优化程序性能。

整个实验不仅提升了我的编程技能，也加深了我对并行计算理论的理解。我学会了如何将理论知识应用到实际问题中，并通过实践来验证和改进我的解决方案。此外，我也认识到了在并行计算中，算法设计、性能优化和工具使用的重要性。