中山大学计算机学院本科生实验报告

（2024 学年秋季学期）

课程名称：高性能计算程序设计基础　　批改人：

实验	应用性能优化	专业（方向）	信息与计算科学
学号	22336049	姓名	陈日康
Email	chenih5@mail2.sysu.edu.cn	完成日期	2025 年 1 月 13 日

1. 实验内容

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。
2. 将 heated_plate_openmp 应用改造成基于 MPI 的进程并行应用。
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。

2. 实验过程和核心代码

(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。

(i). thread_work

```
void* thread_work(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    for (int i = data->start; i < data->end; ++i) {
        data->func(i, data->arg);
    }
    return NULL;
}
```

函数是 Pthreads 线程的执行函数，每个线程在自己的任务范围内调用传入的函数 func。ThreadData 结构体包含：该线程的任务范围 start 和 end。Func 是线程要执行的操作，即 update_solution。线程获取自己的 start 和 end。在 start 到 end 之间，调用 func(i, arg)处理任务。

(ii). matrix_multiplication

```
void parallel_for(int start, int end, void (*func)(int, void*), void* arg) {
    pthread_t threads[NUM_THREADS];
    ThreadData thread_data[NUM_THREADS];
    int chunk_size = (end - start) / NUM_THREADS;

    for (int t = 0; t < NUM_THREADS; ++t) {
        thread_data[t].start = start + t * chunk_size;
        thread_data[t].end = (t == NUM_THREADS - 1) ? end : start + (t + 1) * chunk_size;
        thread_data[t].func = func;
        thread_data[t].arg = arg;
        pthread_create(&threads[t], NULL, thread_work, &thread_data[t]);
    }

    for (int t = 0; t < NUM_THREADS; ++t) {
        pthread_join(threads[t], NULL);
    }
}
```

将一个 for 循环并行化，使用 Pthreads 分配多个线程执行任务。start 和 end 是迭代范围。Func 是在线程中执行的函数。计算任务分块 chunk_size = (end - start) / NUM_THREADS 计算每个线程的工作量。每个线程执行 thread_work，它调用 func 处理 chunk_size 任务。pthread_join 等待所有线程执行完毕。

(2). 将 heated_plate_openmp 应用改造成基于 MPI 的进程并行应用。

(i). MPI 初始化与基本设置

```
// 初始化 MPI 环境
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

// 每个进程分配的行范围
int rows_per_proc = M / size;
int start_row = rank * rows_per_proc;
int end_row = (rank == size - 1) ? M - 2 : start_row + rows_per_proc - 1;
```

MPI_Init 初始化 MPI 环境。MPI_Comm_rank 获取当前进程的进程编号 rank。MPI_Comm_size 获取 MPI 进程的总数。然后计算每个进程负责的行数 rows_per_proc。计算起始行 start_row 和结束行 end_row，确保最后一个进程能正确处理剩余的行。

(ii). MPI 广播初始数据

```
MPI_Bcast(&w, M * N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&mean, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

由主进程广播初始温度网格 w 和初始平均值 mean 给所有进程，确保所有进程从相同的状态开始计算。

(iii). 初始化内部点

```
for (i = start_row; i <= end_row; i++) {
    for (j = 1; j < N - 1; j++) {
        w[i][j] = mean;
    }
}
```

每个进程根据 mean 赋值自己负责的区域（忽略边界）。

(iii). MPI 进程间数据交换

```
if (rank > 0) {
    MPI_Send(&w[start_row][1], N - 2, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD);
    MPI_Recv(&w[start_row - 1][1], N - 2, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
if (rank < size - 1) {
    MPI_Send(&w[end_row][1], N - 2, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD);
    MPI_Recv(&w[end_row + 1][1], N - 2, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

相邻进程交换边界数据，确保计算时 w[i-1][j]和 w[i+1][j]是最新的值：rank > 0 的进程向上发送并接收来自上方的行。rank < size - 1 的进程向下发送并接收来自下方的行。

(iv). MPI 归约计算全局最大变化量

```
// 归约计算全局最大差值
MPI_Allreduce(&local_diff, &diff, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
```

所有进程的 local_diff 取最大值，得到 diff，判断是否满足 EPSILON 停止条件。

(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。

分析不同规模下的并行化应用的执行时间对比；不同规模下的并行化应用的内存消耗对比。“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。内存消耗采用 “valgrind –tool=massif --time-unit=B ./your_exe”工具采集，注意命令 valgrind 命令中增加--stacks=yes 参数采集程序运行栈内内存消耗。Valgrind –tool=massif 输出日志（massif.out.pid）经过 ms_print 打印。

3. 实验结果

(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。

命令格式：`gcc -o heated_plate_pthreads heated_plate_pthreads.c -lpthread -lm`
`./heated_plate_pthreads`

```
HEATED_PLATE_PTHREADS
  Pthreads 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

  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 = 87.978663 seconds
```

(2). 将 heated_plate_openmp 应用改造成基于 MPI 的进程并行应用。

命令格式：chmod +x run_mpi.sh

./run_mpi.sh

```
Compilation successful.
HEATED_PLATE_MPI
  Grid size: 500 x 500
  Number of processes: 4
  Iteration until diff <= 1.000000e-03

 Iteration  Change
         1  18.737475
         2  12.500000
         4  3.126566
         8  1.325619
        16  0.590421
        32  0.280317
        64  0.140488
       128  0.070383
       256  0.035219
       512  0.017614
      1024  0.008810
      2048  0.004391
      4096  0.002196
      8192  0.001110

      9100  0.001000

  Error tolerance achieved.
  Wallclock time: 278.228041 seconds
Execution completed successfully.
```

(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。

命令格式：

```
valgrind --tool=massif --time-unit=B --stacks=yes ./heated_plate_pthreads <进程数> <规模数>
ms_print massif.out.pid
```

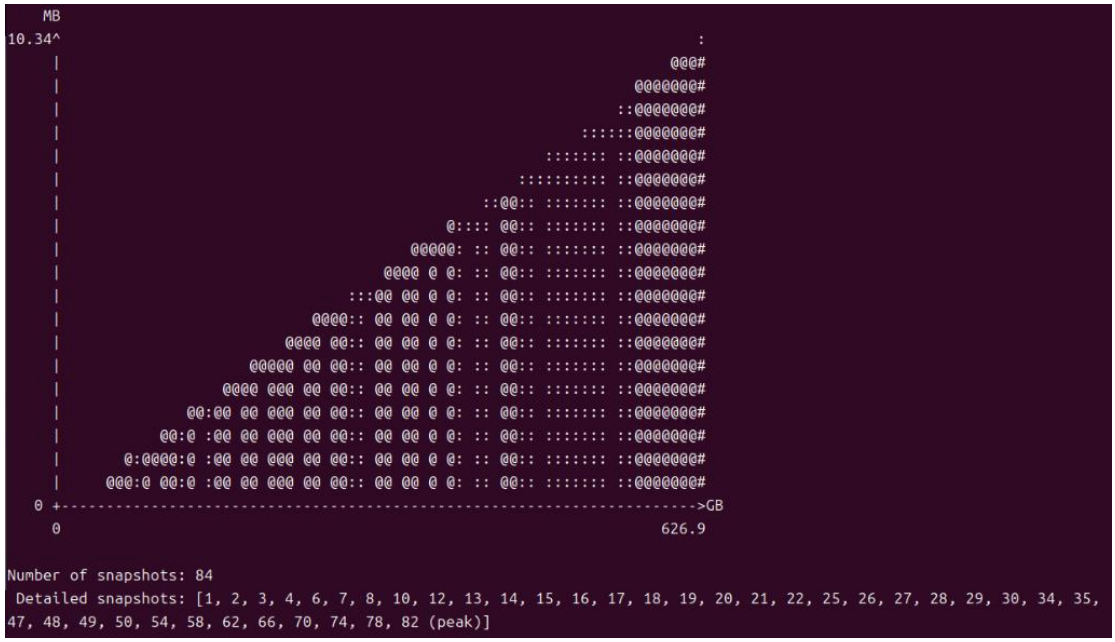
根据定义，问题规模 N 或者 M 的取值范围为 2,4,6,8,16,32,64,…,2097152；并行规模的取值范围为 1,2,4,8 个进程/线程。由于虚拟机仅分配了四个处理器，因此实际测试时进程数选择为 1,2,4，问题规模则选取 16,512,1024 进行实验。接下来，将在不同的并行配置和问题规模下进行测试，以分析并行计算的性能变化趋势，并评估并行加速比、计算效率及其随问题规模变化的影响。

进程数为 1，问题规模为 16：

运行时间：

```
Error tolerance achieved.
Wallclock time = 708.269253 seconds
```

Memory Usage Graph：

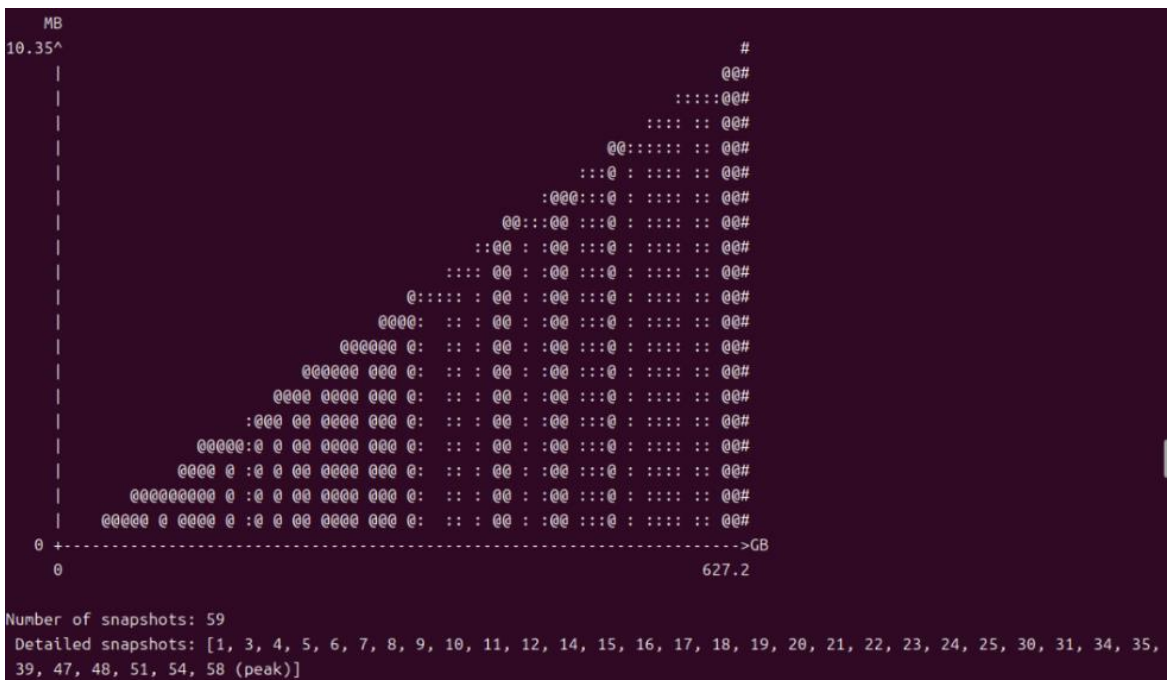


进程数为 1，问题规模为 512：

运行时间：

```
Error tolerance achieved.
Wallclock time = 711.220506 seconds
```

Memory Usage Graph：

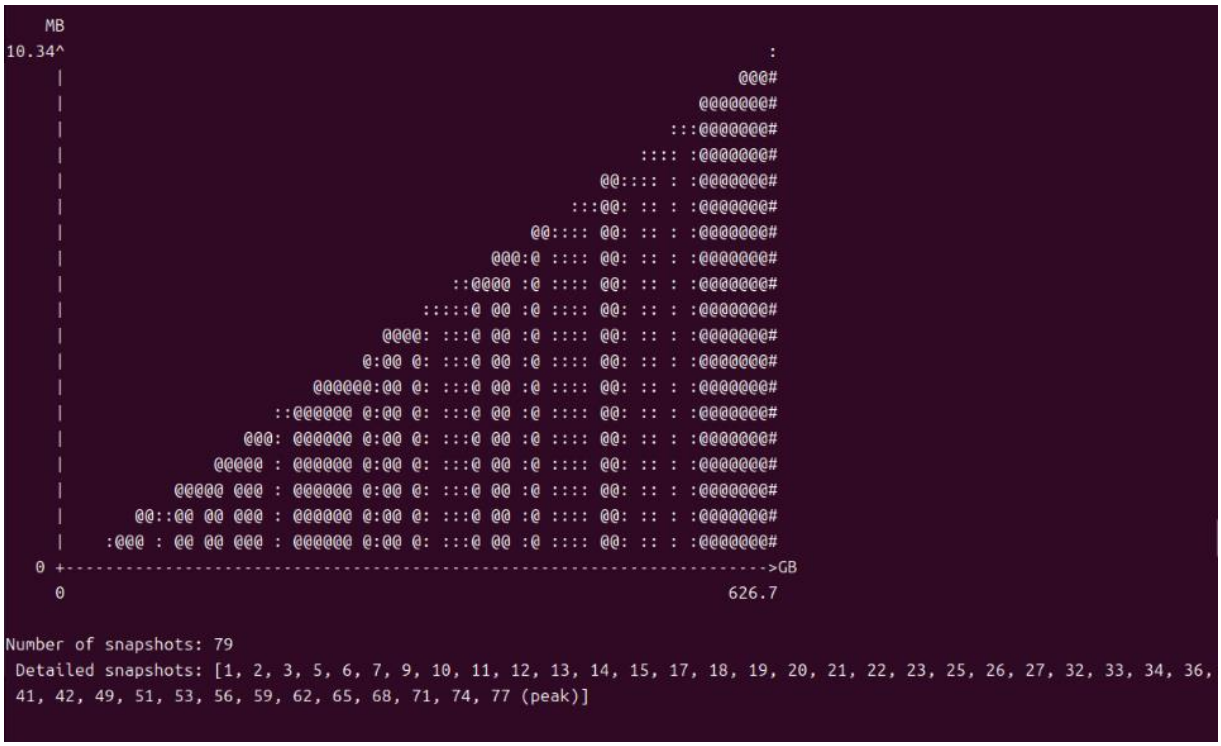


进程数为 1，问题规模为 1024：

运行时间：

```
Error tolerance achieved.
Wallclock time = 709.166584 seconds
```

Memory Usage Graph：

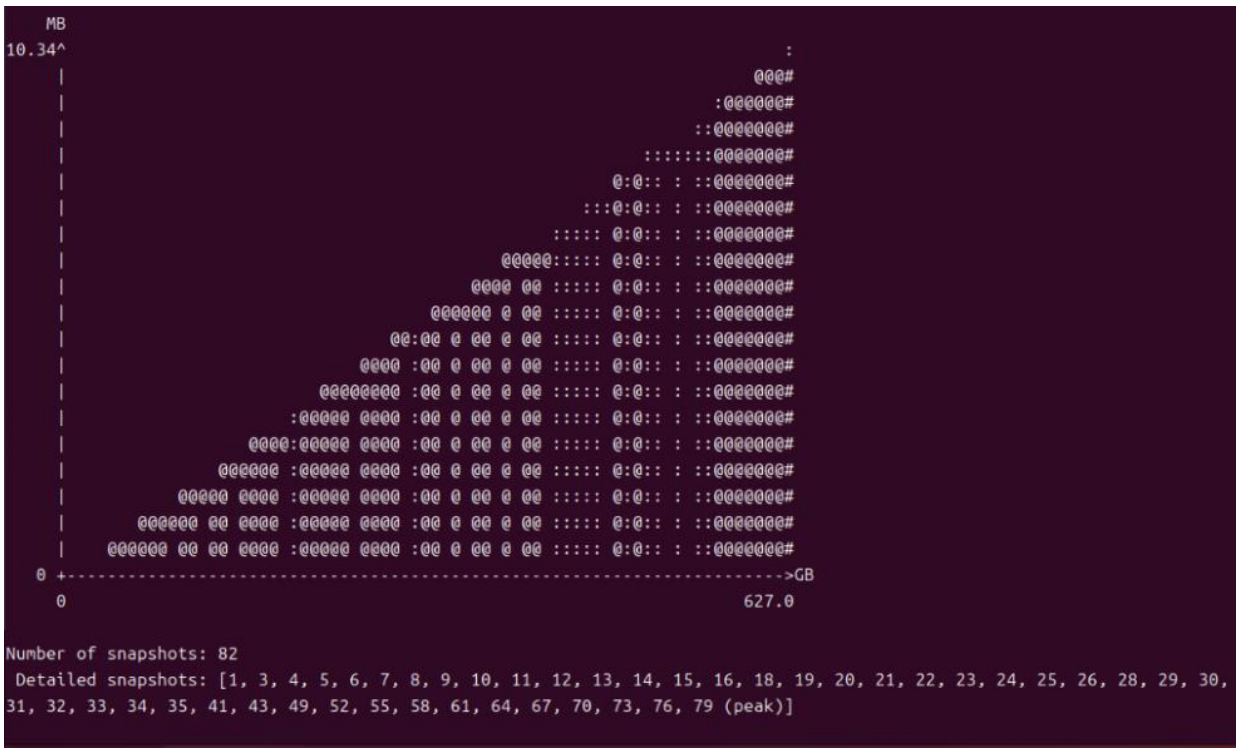


进程数为 2，问题规模为 16：

运行时间：

```
Error tolerance achieved.
Wallclock time = 718.555747 seconds
```

Memory Usage Graph：

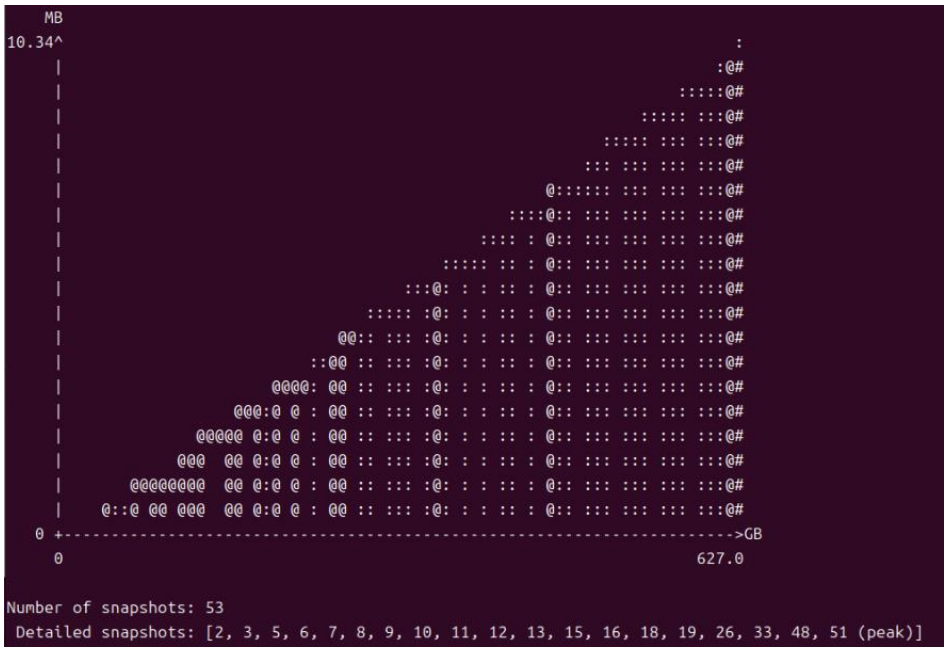


进程数为 2，问题规模为 512：

运行时间：

```
Error tolerance achieved.
Wallclock time = 747.614716 seconds
```

Memory Usage Graph：

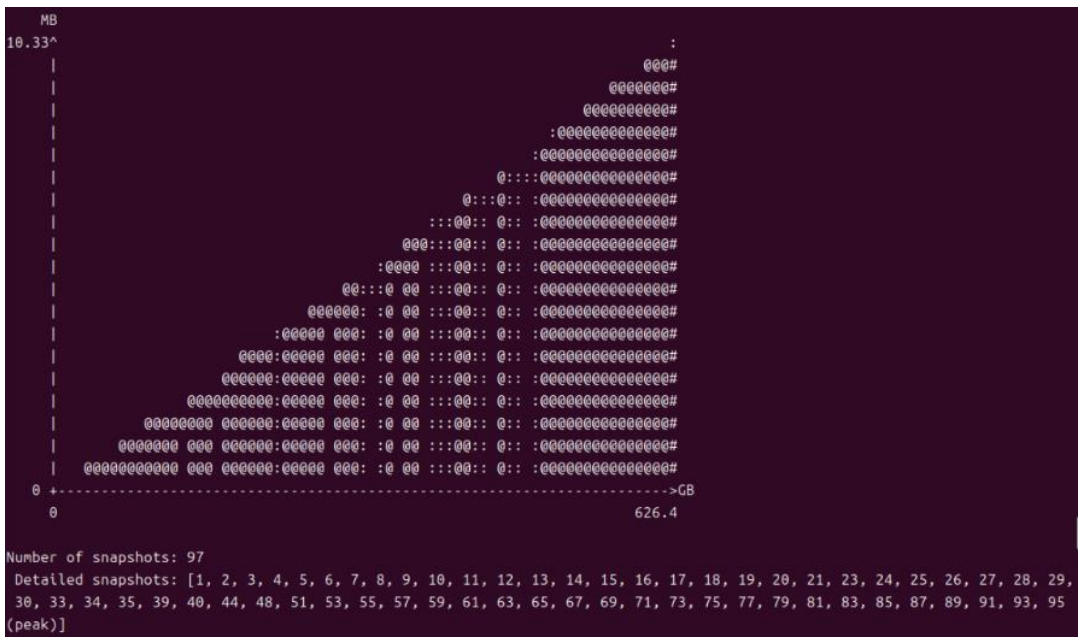


进程数为 2，问题规模为 1024：

运行时间：

```
Error tolerance achieved.
Wallclock time = 741.772174 seconds
```

Memory Usage Graph：

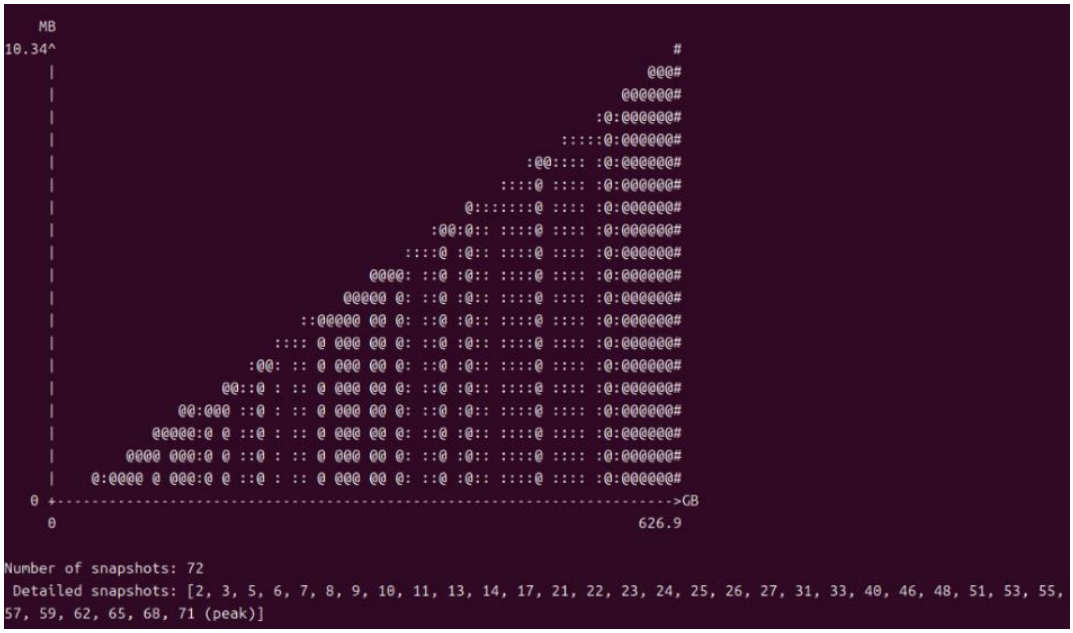


进程数为 4，问题规模为 16：

运行时间：

```
Error tolerance achieved.
Wallclock time = 738.628064 seconds
```

Memory Usage Graph：

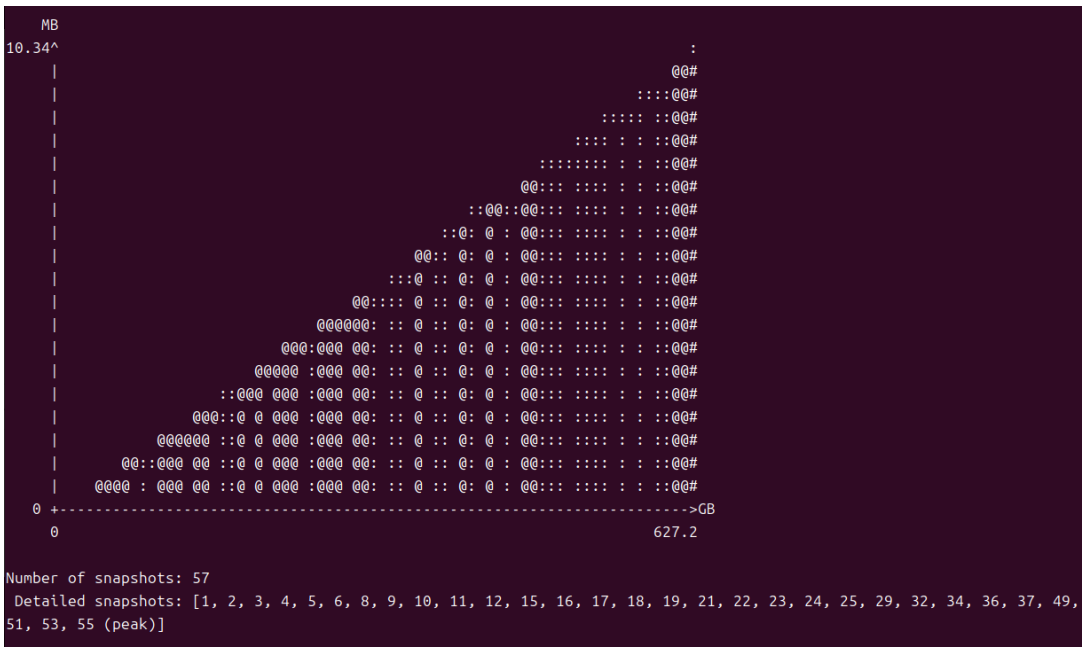


进程数为 4，问题规模为 512：

运行时间：

```
Error tolerance achieved.
Wallclock time = 742.137428 seconds
```

Memory Usage Graph：

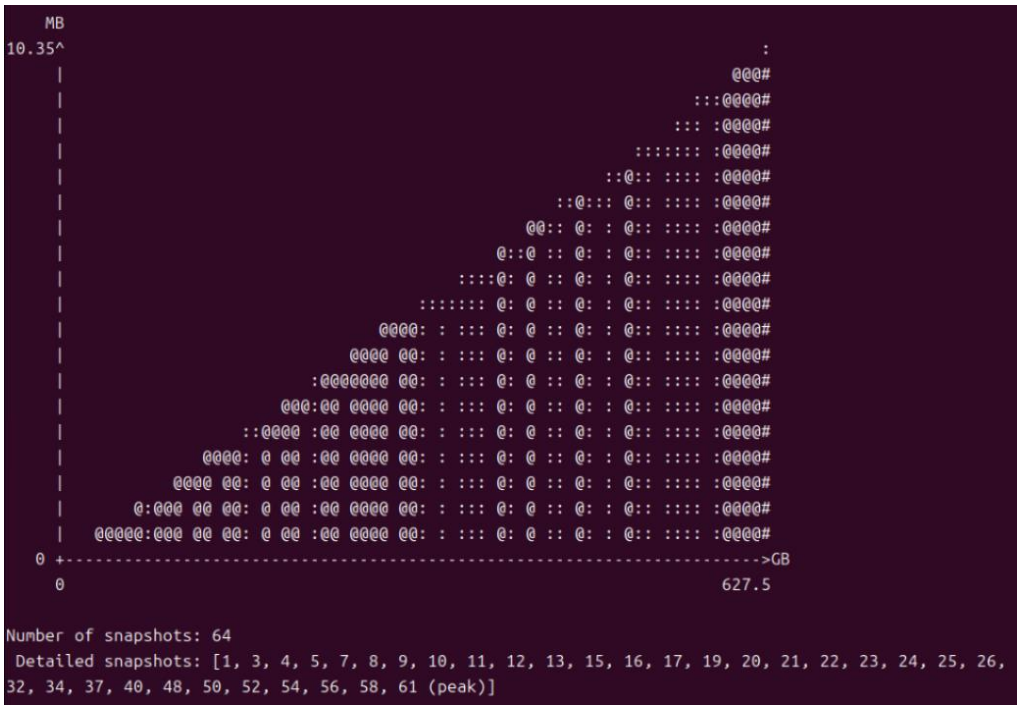


进程数为 4，问题规模为 1024：

运行时间：

```
Error tolerance achieved.
Wallclock time = 726.181291 seconds
```

Memory Usage Graph：



综合实验结果，可以得出以下结论：

(i). 并行计算的加速效果分析：

小规模问题 (N=16, N=512) 并行化收益有限。并行化后，并没有显著的性能提升，甚至随着进程数的增加，运行时间反而有所上升。这表明在小规模任务中，并行计算的通信开销和线程管理成本会抵消并行带来的加速优势。表明在小规模任务下，串行计算可能更具优势。

对于大规模问题 (N=1024) 并行计算的优势逐渐显现，尤其是在 4 线程/进程的情况下，相较于串行版本运行时间明显减少。这是因为线程/进程的计算任务足够大，使得并行计算的计算效率大于管理和通信开销，有效分摊了计算负担。线程数较少时，程序必须依靠串行计算逐步迭代，计算速度受限，而较多的线程能够并行计算多个数据块，从而减少整体迭代步骤，提高计算效率。

(ii). 并行计算的加速效果取决于任务的计算密度和通信开销：

对于小规模问题 (N=16, 512) 内存消耗随着线程/进程数量的增加而增大，主要原因是因为线程/进程需要额外的栈空间进行存储和计算。Pthreads 或进程间通信 MPI 需要额外的缓冲区，导致内存占用增加。由于问题规模较小，计算任务本身对内存的需求并不高，进程数过多时会导致额外的资源消耗，但未能带来明显的性能提升。

对于大规模问题 (N=1024) 线程数较多时，内存占用比串行版本更大，但由于计算效率提升，整体执行时间缩短，因此内存的使用更加有效。进程间通信的影响仍然存在，但并行加速比足够大，弥补了额外的内存开销。在计算密集型任务中，并行计算可以充分发挥性能优势，而在通信密集型任务或小规模计算中，并行化可能带来额外的开销，从而影响性能。

4. 实验感想

本次实验的主要目标是通过 Pthreads 和 MPI 分别对 heated_plate_openmp 进行并行优化，并进行性能分析。用 parallel_for 替换 heated_plate_openmp 中计算量较大的 for 循环，使其能够利用多线程并行执行。随后，又将该应用转换为 MPI 进程并行版本，使其能够在分布式计算环境下运行。

在 MPI 的实验过程中，我了解到进程间通信是影响性能的关键因素。由于每个进程有自己独立的地址空间，因此必须通过 MPI_Send 和 MPI_Recv 进行数据交换，这使得程序的复杂性增加，同时也引入了额外的开销。在较小规模的问题下，通信开销可能比计算时间更大，从而导致加速比不理想。但随着问题规模的增长，MPI 的优势逐渐显现，在未来的高性能计算应用中，MPI 仍然是主流的并行计算框架之一。

通过 valgrind 工具分析内存使用情况，也让我学习到并行程序的内存管理。特别是在 MPI 进程并行中，由于进程的独立性，每个进程都需要独立存储数据，可能导致内存消耗增加。相比之下，Pthreads 在共享内存环境下，可以更高效地利用系统资源，但也需要特别注意竞争条件和同步问题，否则可能会导致数据不一致或性能下降。

总体而言，这次实验不仅帮助我巩固了并行计算的基本概念，还通过实际代码的编写和优化，使我对高性能计算的实现方法和优化策略有了更直观的理解，也对并行程序的优化方向有了更明确的认识。