中山大学数据科学与计算机院本科生实验报告

（2020学年秋季学期）

课程名称：**高性能计算程序设计**　　任课教师：**黄聃**　　批改人：

年级+班级	**18级计7**	**专业（方向）**	**计科**
学号	18340166	**姓名**	王若琪
Email	wangrq29@mail2.sysu.edu.cn	**完成日期**	2020.11.28

1. 实验目的

- 任务 1：

  通过实验 4 构造的基于 Pthreads 的 parallel_for 函数替换 heated_plate_openmp 应用中的“omp parallel for”，实现 for 循环分解、分配和线程并行执行。

- 任务 2：

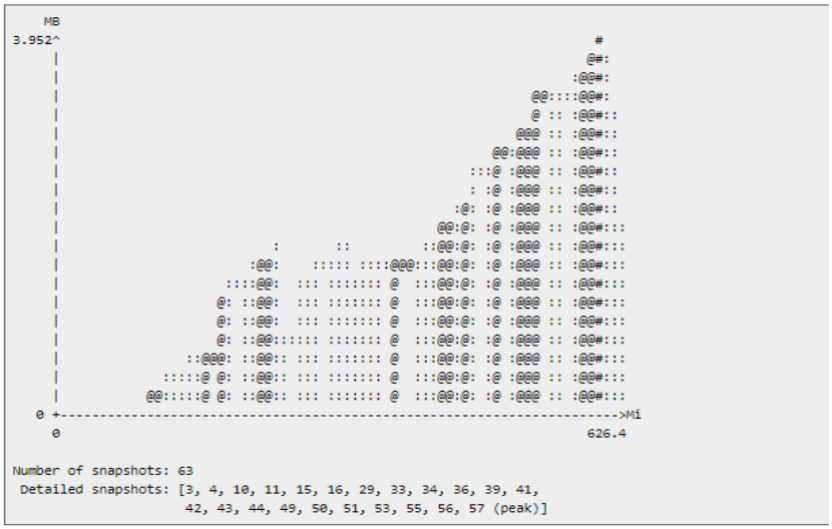
  将heated_plate_openmp应用改造成基于MPI的进程并行应用。 Bonus: 使 用 MPI_Pack/MPI_Unpack ， 或 MPI_Type_create_struct 实现数据重组后的消息传递。

- 任务 3：

  性能分析任务1、任务2和heated_plate_openmp应用， 包括：

  1）不同问题规模的 heated_plate 应用并行执行时间对比，其中问题规模定义为 plate 为正方形，长宽相等， 边长（M=N）变化范围 250，500，1000，2000；并行规模为 1，2，4，8 进/线程。

  2）内存消耗对比，内存消耗采用 “valgrind massif” 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 massif 输出 日志（massif.out.pid）经过 ms_print 打印后示例如下 图，其中 x 轴为程序运行时间，y 轴为内存消耗量：



---

2. 实验过程和核心代码

2.1 改编成 parallel_for 版本

沿用上个实验完成的 parallel_for 函数，此函数代码与上次实验完全相同，故这里不做赘述。

将 heated_plate_openmp.c 代码中的所有omp parallel for改编为函数形式，共10处，举例说明如下：

第一处改编为函数 func_1() 代码如下：

```
void *func_1(void *args)
{
  struct for_index *index = (struct for_index *)args;
  for (int i = index->start; i < index->end; i = i + index->increment)
  {
    w[i][0] = 100.0;
  }
}
```

并在原来的循环的位置调用函数 parallel_for() :

```
parallel_for(1, M - 1, 1, func_1, NULL, ThreadNumber);
```

这样就完成了一个omp parallel for的改编。

第二、三、四、七、八、九、十处改编与第一处改编原理相同，故不做赘述，具体请参考源文件 parallel_for_version.c 。

第五、六处改编应加入互斥变量，保证 mean 的数值的正确性，以第五处为例，改变函数代码如下：

```
void *func_5(void *args)
{
  struct for_index *index = (struct for_index *)args;
  pthread_mutex_lock(&mutex); //获得临界区的访问权
  for (int i = index->start; i < index->end; i = i + index->increment)
  {
    mean = mean + w[i][0] + w[i][N - 1];
  }
  pthread_mutex_unlock(&mutex); //退出临界区
}
```

同样也在原来的循环的位置调用函数 parallel_for() :

```
parallel_for(1, M - 1, 1, func_5, NULL, ThreadNumber);
```

改编完成10处之后，再加入从命令行传参线程数的功能，并将数组改变成动态申请内存的方式，这一任务就完成了，其余细节请参考源文件 parallel_for_version.c 。编译运行指令如下：

```
gcc -std=c99 -g -o parallel_for_version parallel_for_version.c -lpthread
./parallel_for_version <number of threads>
```

2.2 改编成 MPI 版本

先进行初始化 MPI 的众多参数：

```
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

再将对矩阵初始化的几个循环改编为 MPI 版本，关键代码如下：

```
  for (i = 0; i < local_m; i++)
  {
    for (j = 1; j < local_n + 1; j++)
    {
      if (i == 0)
        w[i][j] = 0.0;
      else if (i == local_m - 1)
```

```
        w[i][j] = 100.0;
      else if ((rank == 0) && j == 1)
        w[i][j] = 100.0;
      else if ((rank == size - 1) && j == local_n)
        w[i][j] = 100.0;
      else
        w[i][j] = mean;
    }
  }
```

接下来，改编循环计算直到误差小于0.001的模块，多个线程同时进入循环并判断跳出循环条件：

```
while (diff >= epsilon)
```

在此循环内，先确定各个进程的数据，用 MPI_Sendrecv 进行发送和接受数据：

```
    for (i = 0; i < local_m; i++)
      sendbuf[i] = w[i][1];
    MPI_Sendrecv(sendbuf, local_m, MPI_DOUBLE, L, tag,
                 recvbuf, local_m, MPI_DOUBLE, R, tag,
                 MPI_COMM_WORLD, &status);
    for (i = 0; i < local_m; i++)
      w[i][local_n + 1] = recvbuf[i];
    for (i = 0; i < local_m; i++)
      sendbuf[i] = w[i][local_n];
    MPI_Sendrecv(sendbuf, local_m, MPI_DOUBLE, R, tag,
                 recvbuf, local_m, MPI_DOUBLE, L, tag,
                 MPI_COMM_WORLD, &status);
    for (i = 0; i < local_m; i++)
      w[i][0] = recvbuf[i];
```

保存旧的结果到 u 矩阵的循环改编如下：

```
    for (i = 0; i < local_m; i++)
    {
      for (j = 0; j < local_n + 2; j++)
      {
        u[i][j] = w[i][j];
      }
    }
```

决定新的点的结果的双层循环改编如下，在开始时先进行对不同进程中 start_col 和 end_col 的分类讨论：

```
for (i = 1; i < local_m - 1; i++)
{
  if (rank == 0)
  {
    start_col = 2;
    end_col = local_n;
  }
  else if (rank == size - 1)
  {
    start_col = 1;
    end_col = local_n - 1;
  }
  else
  {
    start_col = 1;
    end_col = local_n;
  }
  for (j = start_col; j < end_col + 1; j++)
  {
    w[i][j] = (u[i - 1][j] + u[i + 1][j] + u[i][j - 1] + u[i][j + 1]) / 4.0;
  }
}
```

同理，计算diff的双层循环改编如下，在开始时先进行对不同进程中 start_col 和 end_col 的分类讨论：

```
for (i = 1; i < local_m - 1; i++)
{
  if (rank == 0)
  {
    start_col = 2;
    end_col = local_n;
  }
  else if (rank == size - 1)
  {
    start_col = 1;
    end_col = local_n - 1;
  }
  else
  {
    start_col = 1;
    end_col = local_n;
  }
  for (j = start_col; j < end_col + 1; j++)
  {
```

```
            if (diff < fabs(w[i][j] - u[i][j]))
            {
              diff = fabs(w[i][j] - u[i][j]);
            }
          }
        }
```

将多个进程算好的diff发送给主进程，并在主进程中进行比较，取得最大值：

```
    if (rank == 0)
    {
      int i;
      double temp_diff = 999;
      for (i = 1; i < size; i++)
      {
        MPI_Recv(&temp_diff, 1, MPI_DOUBLE, i, 20, MPI_COMM_WORLD, &status);
        if (temp_diff > diff)
          diff = temp_diff;
      }
    }
    if (rank != 0)
    {
      MPI_Send(&diff, 1, MPI_DOUBLE, 0, 20, MPI_COMM_WORLD);
    }
```

最后一定要注意用 MPI_Bcast 函数对得到的 diff 值进行广播，否则程序将会陷入死锁：

```
    MPI_Bcast(&diff, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

2.3 将 OpenMp 版本的改编为可设定线程数的方式

在每一句相关并行指令的后面加上 num_threads(ThreadNumber)，如：

```
#pragma omp parallel shared(w) private(i, j) num_threads(ThreadNumber)
```

再将二维数组设定为能够进行动态分配内存的形式，便于增大 M、N 后的实验。

2.4 性能分析

- 性能对比：

  对三种方式进行不同线程数和不同规模的横向、纵向对比，具体运行时间对比分析见结果部分。

- 内存消耗对比：

  内存消耗采用 “valgrind massif” 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，例如：

  ```
  valgrind --tool=massif --stacks=yes ./heated_plate_openmp 8
  ms_print massif.out.3896
  ```

  ```
  valgrind --tool=massif --stacks=yes ./parallel_for_version 8
   ms_print massif.out.5641
  ```

  ```
  valgrind --tool=massif --stacks=yes mpirun -np 8 mpi_version
  ms_print massif.out.23269
  ```

  具体结果图像和分析见结果部分。

---

3. 实验结果

3.1 不同规模运行时间对比：

M=N=250 时运行时间对比如下表：

线程数\并行方式	parallel_for	MPI	OpenMp
1	12.161629 s	5.438418 s	5.495690 s
2	4.632318 s	2.862456 s	2.944783 s
4	4.965635 s	1.703922 s	1.580619 s
8	4.579512 s	1.745327 s	1.493191 s

M=N=500 时运行时间对比如下表：

线程数\并行方式	parallel_for	MPI	OpenMp
1	79.117024 s	46.173493 s	49.046285 s
2	45.388273 s	29.476897 s	29.442187 s
4	42.030196 s	20.647842 s	19.153715 s
8	35.296184 s	21.939559 s	19.396372 s

M=N=1000 时运行时间对比如下表：

线程数\并行方式	parallel_for	MPI	OpenMp
1	271.957345 s	236.029441 s	254.477902 s
2	172.908708 s	158.532431 s	164.758838 s
4	153.912302 s	104.223597 s	104.728386 s
8	139.602400 s	118.241375 s	105.800258 s

M=N=2000 时运行时间对比如下表：

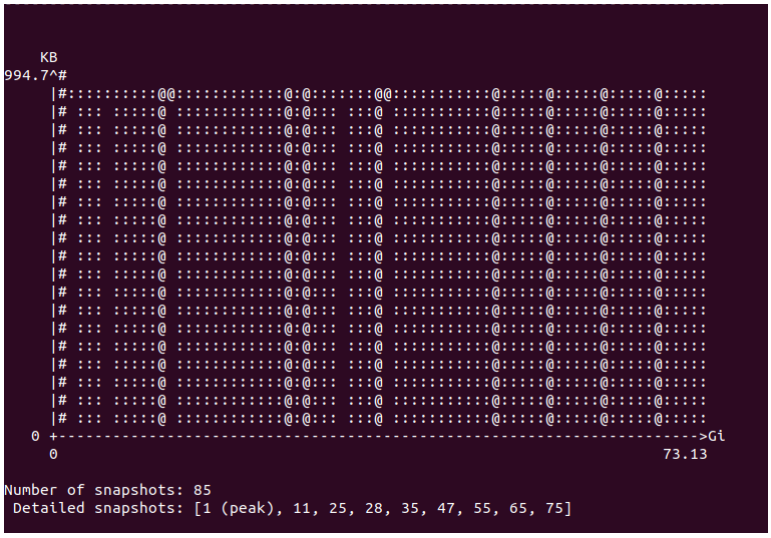
线程数\并行方式	parallel_for	MPI	OpenMp
1	1049.004547 s	970.929837 s	1065.631812 s
2	645.989642 s	444.759709 s	713.198161 s
4	607.012345 s	403.636525 s	431.536380 s
8	442.455684 s	295.611280 s	469.753905 s

通过以上四个表格，观察可得，在矩阵规模较小时（250、 500、 1000），改编之后的 MPI 版本和 OpenMp 版本运行时间很接近，性能相似，而 parallel_for 版本的运行时间明显比其他两种方式长，可见其性能不如其他两种方式。在矩阵规模较大时（2000），改编之后的 MPI 版本性能最好，运行时间最短，parallel_for 版本略优与 OpenMp 版本，但差距不大。

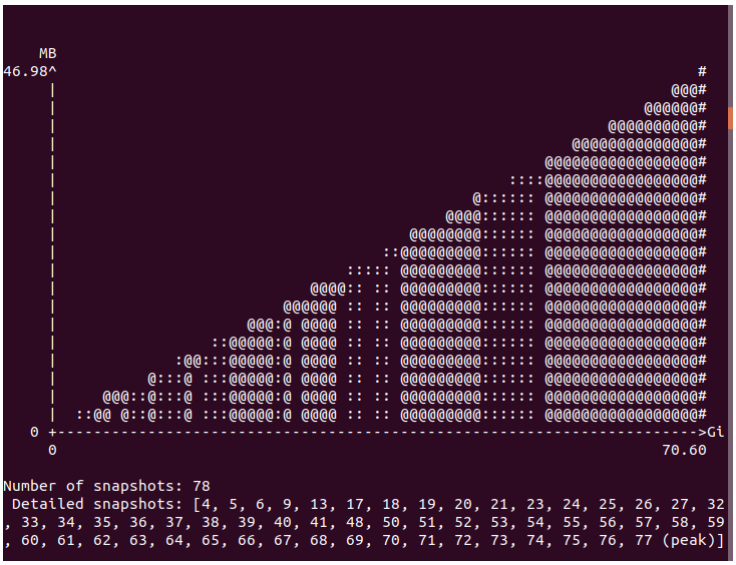
3.2 内存消耗对比

以 250×250 的规模，8 线程运行为例，对比三种不同并行方式的内存消耗：

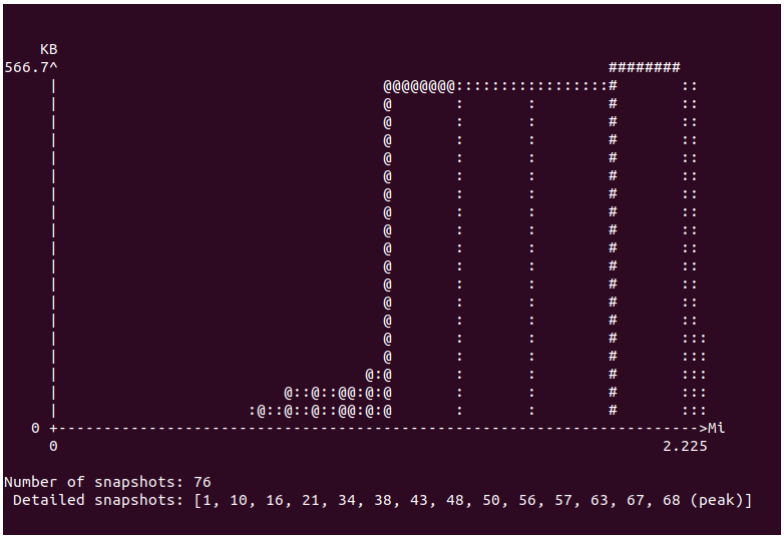
OpenMp版本内存消耗图像：



Parallel_for 版本内存消耗图像：



MPI 版本内存消耗图像：



通过以上三幅图像对比可以看出，OpenMp版本内存消耗图像随时间的图像较为均衡；Parallel_for 版本的随时间内存占用逐渐增大，并且占用内存是三种并行方式中最多的；MPI 并行版本占用内存相对较少。

4. 实验感想

- 在做任务一时，我刚开始遇到了改编后的版本运行速度明显慢于原版本的问题，经检查，我发现原来是因为我的第十处更改多加了互斥变量机制，但实际上不需要加，所以经过修改后，解决了运行速度慢的问题。
- 在将 heated_plate_openmp.c 改编成能够决定线程数的模式时，遇到了 num_threads() 加错位置的错误，刚开始不知道要在外面的大括号那层加，就加到了内层，造成了报错，后来经查阅资料，发现可以加到外层，这样能达到对内侧所有循环同时限定线程数的效果。
- 改编为 MPI 版本是本实验中最难完成的一部分，开始时准备每个循环都改编一次，后来参考了网上的资料，将很多个循环结合在一起改编，省了很多代码量和时间。在 MPI 版本改编的时候，开始遇到了不能在精度达到要求时停止的现象，造成死锁，后来发现是因为没有将 diff 广播到每个进程，所以添加了 Bcast 语句，就正确了。
- 通过性能的对比，我发现自己编写的 parallel_for 版本的性能在矩阵规模较小时，性能明显不如其他两个版本，但是当矩阵规模变大的时候，其性能会略优于 OpenMp 原版，这可能是因为 parallel_for 的固有开销比较大，造成矩阵规模小的时候有较多的线程开销，导致时间变长。
- 第一次使用“valgrind massif” 工具采集并观察分析程序的内存消耗，感到很有用，之后可以多尝试使用这样的工具来辅助我写出更好的程序。
