中山大学数据科学与计算机院本科生实验报告

(2020 秋季学期)

课程名称：高性能程序设计　**任课老师**：黄聃　**批改人**：

年级 + 班级	18 级计算机	年级 (方向)	计算机科学
学号	18340236	姓名	朱煜
Email	zhuy85@mail2.sysu.edu.cn	完成日期	2020.12.02

1 实验目的

1.1 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。

1.2 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。

1.3 任务 3

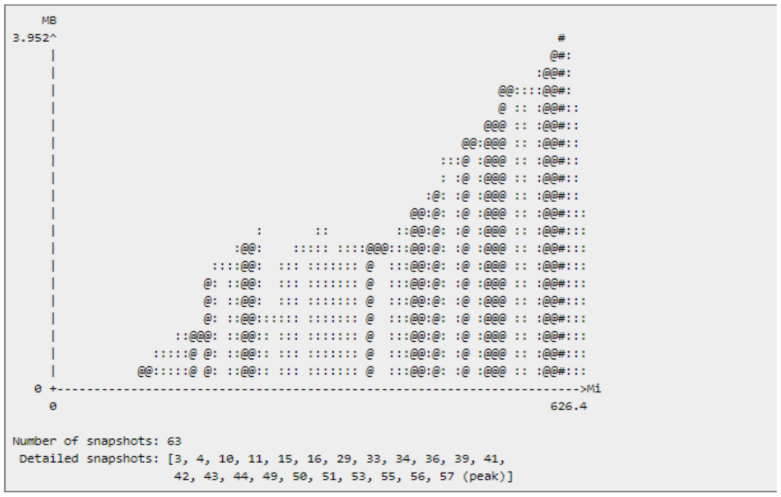
性能分析任务 1、任务 2 和 heated_plate_openmp 应用，包括：

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 500，1000，2000，4000；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：

1.4 参考文献：

Valgrind massif
Heated_plate_openmp 源代码
Heated_plate_openmp 介绍



2 实验过程

2.1 任务 1

根据实验 4 实现的基于 Pthreads 的 parallel_for 函数，函数的参数需按照结构体指针的方式传入，将各线程的共享变量按指针的方式保存在结构体中，定义的结构体代码如下

```
struct args
{
  double (*w)[N];
  double *mean;
  double (*u)[N];
  double *diff;
  args(double (*tw)[N], double *tmean, double (*tu)[N], double *tdiff)
  {
    w = tw;
    mean = tmean;
    u = tu;
    diff = tdiff;
  }
};
```

按照 parallel_for 函数的定义，对原有可并行的循环进行改写，定义如下。

```
void parallel_for(int start, int end, int crement, void *(*functor)(void
↪  *), void *arg, int num_threads);
```

以第一个 for 循环为例，如下，需要传入循环的开始，结束与递增量。而由于我们将二维数组以指针数组的形式用结构体传入到函数中，则可以从传入的结构体中重新获得所需的全局变量 w。

```
#pragma omp for
    for ( i = 1; i < M - 1; i++ )
    {
      w[i][0] = 100.0;
    }
```

修改后的函数代码如下

```
void *functor0(void *arg)
{
  struct for_index_arg *index = (struct for_index_arg *)arg;
  struct args *true_arg = (struct args *)(index->args);
  for (int i = index->start; i < index->end; i = i + index->increment)
  {
    (true_arg->w)[i][0] = 100.0;
  }
  return NULL;
};
```

在主函数中，我们便可以使用实验 4 的 parallel_for 函数代替原来的 OpenMP 的并行。修改所有的可并行部分使用 parallel_for 函数替换。由于实验中对 w 矩阵与 u 矩阵的划分为按行划分，各线程不存在 race condition 的情况。但在对 $diff$ 变量进行赋值时，需要考虑线程的互斥，在实验中采用了信号量的方式实现对临界区的访问，具体实现代码如下：

```
void *functor9(void *arg)
{
  struct for_index_arg *index = (struct for_index_arg *)arg;
  struct args *true_arg = (struct args *)(index->args);
```

```
5   double temp_diff = 0.0;
6   for (int i = index->start; i < index->end; i = i + index->increment)
7   {
8     for (int j = 1; j < N - 1; j++)
9     {
10      if (temp_diff < fabs((true_arg->w)[i][j] - (true_arg->u)[i][j]))
11      {
12        temp_diff = fabs((true_arg->w)[i][j] - (true_arg->u)[i][j]);
13      }
14    }
15  }
16  // 临界区的访问
17  pthread_mutex_lock(&mutex);
18  if (*(true_arg->diff) < temp_diff)
19  {
20    *(true_arg->diff) = temp_diff;
21  }
22  pthread_mutex_unlock(&mutex);
23  return NULL;
24 };
```

其余关于时间修改的函数不做赘述，具体见源码文件"heated_plate_ParallerFor.cpp"。修改后运行结果如图2，可见采用 parallel_for 函数替换后的并行效果与实验 4 一样差。

2.2 任务 2

与任务 1 相同，采用 MPI 并行时，对 w 矩阵与 u 矩阵的划分为按行划分。每一进程负责其所划分矩阵的初始化与计算，以 4 进程为例 w 矩阵与 u 矩阵划分方式如图3。将 w 矩阵与 u 矩阵平均划分给各线程，当不能整除时，剩余部分矩阵划分给最后一个进程。根据源代码文件 heated_plate_openmp.cpp 的注释，w 矩阵的初始化如图1。

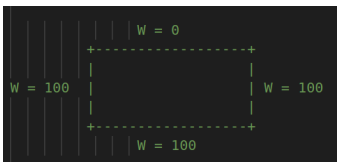


图 1: w 矩阵的初始化

```
HEATED_PLATE_PARALLERFOR
  C/ParallerFor 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 = 35.232529

HEATED_PLATE_PARALLERFOR:
  Normal end of execution.
```

图 2: heated_plate_ParallerFor 运行结果

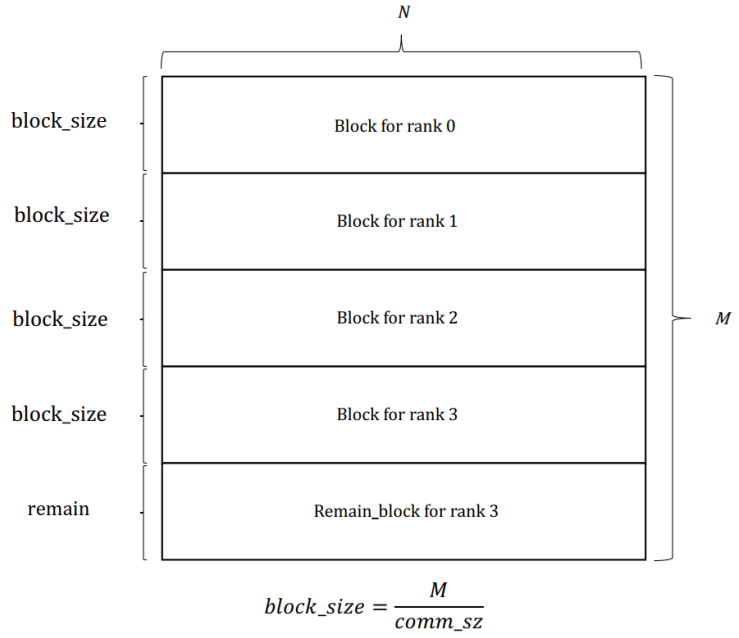


图 3: w 矩阵与 u 矩阵的划分

根据该初始化方式，w 矩阵的第一行初始化为 0，第 $M-1$ 行初始化为 100.0，其余行的第一个元素与末尾元素初始化为 100.0。根据划分的块，使用 MPI 对矩阵进行并行初始化。实现代码如下，分三种情况讨论，这样讨论的方法是我本次实现 MPI 的主要思想，则是对不同线程的执行分别进行考虑。

```
1  if (my_rank == 0)
2  {
3    for (i = for_start; i < for_end; ++i)
4    {
5      for (j = 0; j < N; ++j)
6      {
7        // 第一行全初始化为 0
8        if (i == 0)
9          w[i][j] = 0.0;
10       // 最后一行全初始化为 100.0
11       else if (i == M - 1)
12       {
13         w[i][j] = 100.0;
14         my_mean += w[i][j];
15       }
16       // 中间的行只有初始与末尾元素初始化为 100.0
17       else if (j == 0 || j == N - 1)
18       {
19         w[i][j] = 100.0;
20         my_mean += w[i][j];
21       }
22     }
23   }
24 }
25 else if (my_rank == (comm_sz - 1))
26 {
27   for (i = for_start; i < for_end; ++i)
28   {
29     for (j = 0; j < N; ++j)
30     {
31       // 最后一行全初始化为 100.0
```

```
32          if (i == M - 1)
33          {
34            w[i][j] = 100.0;
35            my_mean += w[i][j];
36          }
37          // 中间的行只有初始与末尾元素初始化为 100.0
38          else if (j == 0 || j == N - 1)
39          {
40            w[i][j] = 100.0;
41            my_mean += w[i][j];
42          }
43        }
44      }
45    }
46    else
47    {
48      // 中间的行只有初始与末尾元素初始化为 100.0
49      for (i = for_start; i < for_end; ++i)
50      {
51        w[i][0] = 100.0;
52        w[i][N - 1] = 100.0;
53        my_mean += (w[i][0] + w[i][N - 1]);
54      }
55    }
```

在上述初始化过程中，各进程同时进行了 $mean$ 值的计算，得到了各自的 my_mean。为了计算总的 $mean$ 值，需要进行进程通信，使用 **MPI_Send** 函数与 **MPI_Recv** 函数将 my_mean 发送给主进程，再由主进程计算出 $mean$ 值后，发送给各进程。在实验中尝试了 **MPI_Pack** 函数与 **MPI_Unpack** 函数。具体实现代码如下：

```
1   if (my_rank == 0)
2   {
3     mean += my_mean;
4     for (i = 1; i < comm_sz; ++i)
5     {
6       // 接收来自其他进程的 mean
```

```
      MPI_Recv(buffer, 100, MPI_PACKED, i, 0, MPI_COMM_WORLD,
      ↪ MPI_STATUS_IGNORE);
      position = 0;
      // 按位置 unpack 到 my_mean 中
      MPI_Unpack(buffer, 100, &position, &my_mean, 1, MPI_DOUBLE,
      ↪ MPI_COMM_WORLD);
      // 累加
      mean += my_mean;
    }
    // 计算总的平均值
    mean = mean / (double)(2 * M + 2 * N - 4);
    printf("\n");
    printf("  MEAN = %f\n", mean);
    for (i = 1; i < comm_sz; ++i)
    {
      // 将计算的结果发送给其他进程
      MPI_Pack(&mean, 1, MPI_DOUBLE, buffer, 100, &position,
      ↪ MPI_COMM_WORLD);
      MPI_Send(buffer, position, MPI_PACKED, i, 1, MPI_COMM_WORLD);
    }
  }
  else
  {
    // 打包 my_mean 给主进程
    MPI_Pack(&my_mean, 1, MPI_DOUBLE, buffer, 100, &position,
    ↪ MPI_COMM_WORLD);
    // 发送给主进程
    MPI_Send(buffer, position, MPI_PACKED, 0, 0, MPI_COMM_WORLD);
    // 接受来自主进程的 mean
    MPI_Recv(buffer, 100, MPI_PACKED, 0, 1, MPI_COMM_WORLD,
    ↪ MPI_STATUS_IGNORE);
    MPI_Unpack(buffer, 100, &position, &mean, 1, MPI_DOUBLE,
    ↪ MPI_COMM_WORLD);
  }
```

之后关于各进程初始化 w 的方式便简单地采用分情况讨论的方式，详细见源码。在实

验中，u 矩阵的划分方式与 w 矩阵相同，在源程序的大部分计算中是可以简单地按块划分。而在并行处理以下代码时会违背采用的分块，可以看到，由于在 u 矩阵第一维索引中出现了 $i-1$ 与 $i+1$，导致各进程会读取其他进程所划分的矩阵块，因此需要相邻块进程将所需的行发送给当前进程。以 4 进程为例，具体过程如图4。每个进程需要其块相邻的行进行计算，如 rank0 块需要 rank1 块的第一行，rank1 块需要 rank0 块的最后一行与 rank2 块的一行。

```
1     # pragma omp for
2           for ( i = 1; i < M - 1; i++ )
3           {
4             for ( j = 1; j < N - 1; j++ )
5             {
6               w[i][j] = ( u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] ) /
                  ↪  4.0;
7             }
8           }
9         }
```

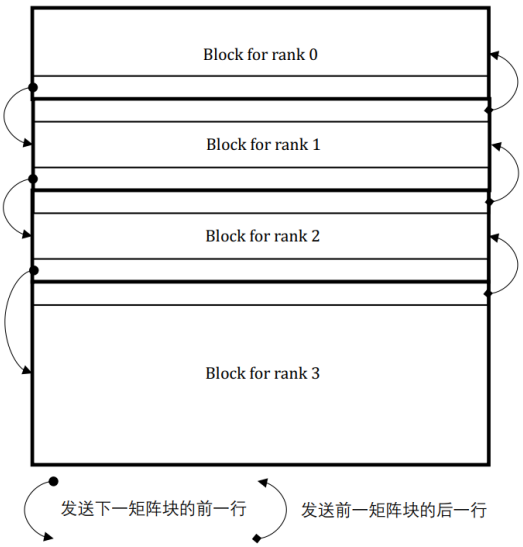


图 4: 进程间行的发送

根据行数发送之间的关系，实现代码如下，按顺序接收传递后，每个进程都可接收到自己计算时所需要的行，并保存到自己的 u 矩阵中。

```
      if (my_rank == 0)
      {
        // 第一块先向下传递
        position = 0;
        MPI_Pack(&w[for_end - 1][0], N, MPI_DOUBLE, buffer, sizeof(double)
        ↪  * (N + 1), &position, MPI_COMM_WORLD);
        MPI_Send(buffer, position, MPI_PACKED, 1, 1, MPI_COMM_WORLD);
        // 再从下接收
        position = 0;
        MPI_Recv(buffer, sizeof(double) * (N + 1), MPI_PACKED, 1, 2,
        ↪  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Unpack(buffer, sizeof(double) * (N + 1), &position,
        ↪  &u[for_end][0], N, MPI_DOUBLE, MPI_COMM_WORLD);
      }
      else if (my_rank == (comm_sz - 1))
      {
        // 最后一块先从上接收
        position = 0;
        MPI_Recv(buffer, sizeof(double) * (N + 1), MPI_PACKED, my_rank - 1,
        ↪  1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Unpack(buffer, sizeof(double) * (N + 1), &position,
        ↪  &u[for_start - 1][0], N, MPI_DOUBLE, MPI_COMM_WORLD);
        // 再向上传递
        position = 0;
        MPI_Pack(&w[for_start][0], N, MPI_DOUBLE, buffer, sizeof(double) *
        ↪  (N + 1), &position, MPI_COMM_WORLD);
        MPI_Send(buffer, position, MPI_PACKED, my_rank - 1, 2,
        ↪  MPI_COMM_WORLD);
      }
      else
      {
        // 中间块先从上接收
        position = 0;
        MPI_Recv(buffer, sizeof(double) * (N + 1), MPI_PACKED, my_rank - 1,
        ↪  1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
28          MPI_Unpack(buffer, sizeof(double) * (N + 1), &position,
            ↪  &u[for_start - 1][0], N, MPI_DOUBLE, MPI_COMM_WORLD);
29          // 再向下传递
30          position = 0;
31          MPI_Pack(&w[for_end - 1][0], N, MPI_DOUBLE, buffer, sizeof(double)
            ↪  * (N + 1), &position, MPI_COMM_WORLD);
32          MPI_Send(buffer, position, MPI_PACKED, my_rank + 1, 1,
            ↪  MPI_COMM_WORLD);
33          // 再再从下接收
34          position = 0;
35          MPI_Recv(buffer, sizeof(double) * (N + 1), MPI_PACKED, my_rank + 1,
            ↪  2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
36          MPI_Unpack(buffer, sizeof(double) * (N + 1), &position,
            ↪  &u[for_end][0], N, MPI_DOUBLE, MPI_COMM_WORLD);
37          // 最后向上传递
38          position = 0;
39          MPI_Pack(&w[for_start][0], N, MPI_DOUBLE, buffer, sizeof(double) *
            ↪  (N + 1), &position, MPI_COMM_WORLD);
40          MPI_Send(buffer, position, MPI_PACKED, my_rank - 1, 2,
            ↪  MPI_COMM_WORLD);
41        }
42      }
```

解决了以上问题后，剩下的并行实现与以上几种实现方式类似，不做赘述，具体见源码文件”heated_plate_MPI.cpp”。修改后运行结果如图5，可见改写后的程序变快了不少，具体比较见任务 3。

2.3 任务 3

2.3.1 并行执行时间对比

调节线程数与矩阵规模进行输入，对实现的任务 1、任务 2 和 heated_plate_openmp 应用测试运算时间，结果（单位：s）如表1，表2，表3。由于采用的虚拟环境只支持 4 线程，8 进程效果如 4 进程类似，没有参考价值。

以上表格数据绘制成折线图结果如图6，可见任务 2 中实现的 MPI 并行效果与原有的 heated_plate_openmp 应用实现效果相近，考虑在 MPI 实现过程中进行了大量的进程通信，在 w 矩阵发送时，由于需要按顺序逐块进行发送，导致其他进程会阻塞在 MPI_Recv

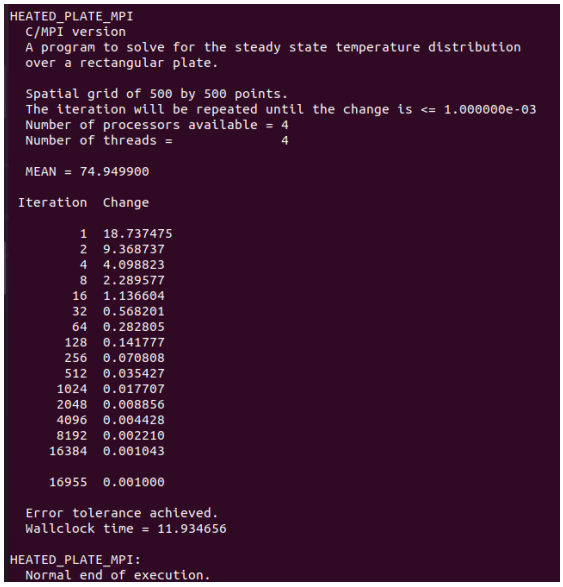


图 5: heated_plate_MPI 运行结果

表 1: **任务 1** 运算时间与线程数、矩阵规模的关系

线程数 \ 矩阵规模	**250**	**500**	**1000**	**2000**
1	20.788555	85.662582	276.841309	950.904114
2	16.747892	49.919853	166.045090	536.490173
4	11.961040	34.892178	108.831253	381.332184

表 2: **任务 2** 运算时间与线程数、矩阵规模的关系

线程数 \ 矩阵规模	**250**	**500**	**1000**	**2000**
1	6.829900	42.652457	187.681650	745.896982
2	3.603191	24.042757	96.963942	408.552895
4	2.030146	14.076491	70.3766	270.969024

表 3: **heated_plate_openmp** 运算时间与线程数、矩阵规模的关系

线程数 \ 矩阵规模	**250**	**500**	**1000**	**2000**
1	7.191927	44.605637	187.175752	750.426360
2	3.750896	25.223615	97.914136	383.318972
4	1.870987	13.308963	62.972270	286.588529

上，这样会浪费通讯时间，在本次实验看来，MPI 的并行效果应该是好于 OpenMP。而任务 1 实现的 parallel_for 性能较差主要原因在于传入参数采用的是指针，在访问参数的时候进行了指针的寻址，而且在 parallel_for 函数中会进行 pthread_t 内存的申请、线程的创建与释放，在实验中每次迭代都需要进行一次这样的创建与释放，消耗了大量时间。

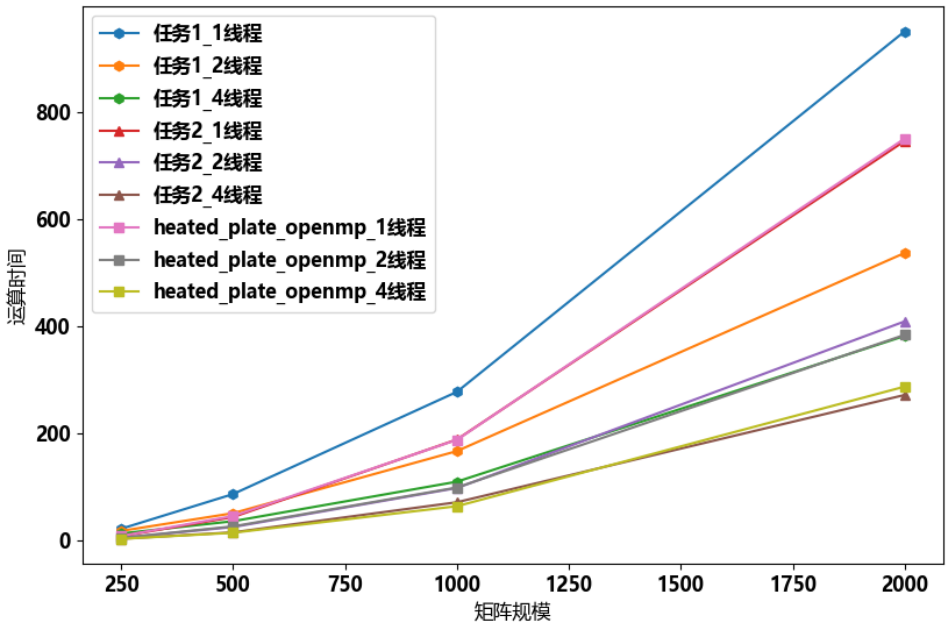


图 6: 运算时间比较

2.3.2 内存消耗对比

采用 **valgrind massif** 工具采集内存消耗，并在命令中增加–stacks=yes 参数采集程序运行栈内内存消耗。经查阅，**massif-visualizer** 可以可视化地展示内存分配随着采样时间的变化情况，并能直观的看到内存分配的排行榜。比实验要求中使用的 ms_print 更加直观，因此选择使用 **massif-visualizer** 进行内存分析的展示。

依次采集三个程序的内存消耗，heated_plate_parallerFor 的内存消耗如图7，由于 parallerFor 函数是在每次并行时进行线程的创建与释放，因此在每次迭代都会使用 malloc 在堆上申请内存。图中可见内存消耗不断增加，考虑可能发生了内存泄漏，在检查了源码后并没有发现 malloc 函数与 free 函数或 new 函数与 delete 函数不配对的情况，而且每次创建进程后都使用 pthread_join 函数释放各进程的占有的资源，最终并没有找到内存泄漏的原因。

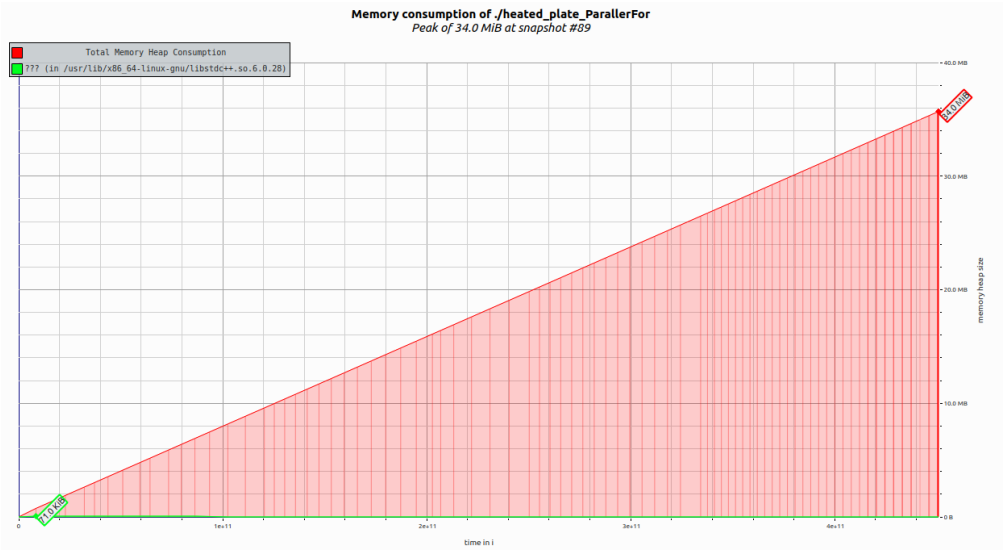


图 7: heated_plate_parallerFor 的内存消耗

heated_plate_MPI 的内存消耗如图8，由于 MPI 中每个进程有自己的内存和变量，这样不用担心冲突问题，但也因此消耗了大量的内存。由于在计算热传递时，进程之间需要进行大量的通讯，使用了 MPI_Send 与 MPI_Recv 函数进行进程通讯，在这个过程中需要缓冲区进行保存，消耗了一定的内存。而最终峰值的内存消耗为 2.4MB，属于正常范围。

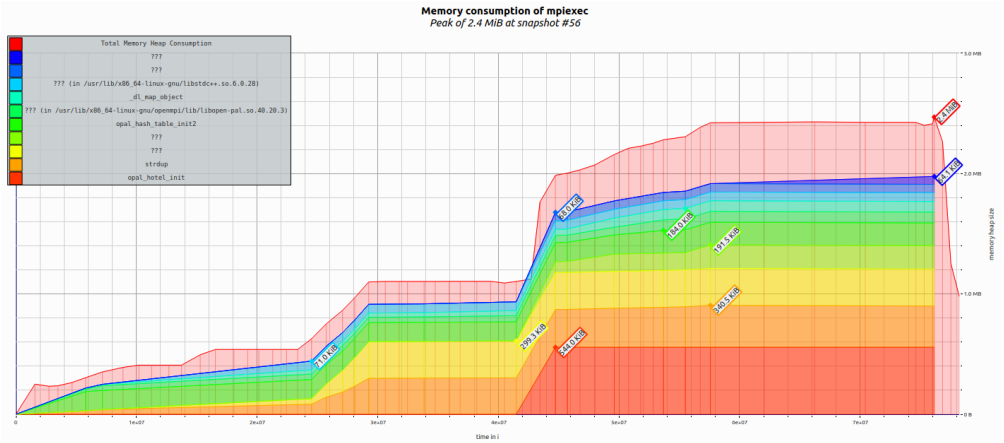


图 8: heated_plate_MPI 的内存消耗

heated_plate_openmp 的内存消耗如图9，由于 openmp 使用线程间共享内存的方式协调并行计算，因此只需在程序的最开始进行变量的声明，在中间基本不会增加新的内存消耗，内存开销小。而 OpenMP 目前主要针对循环并行化，实验中需要对大量的的循环进行

并行处理，因此在本次实验中效果最佳。

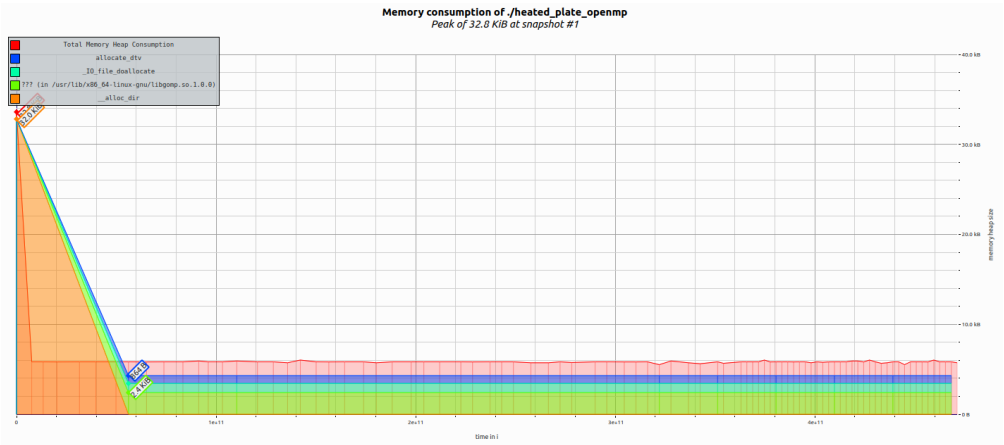


图 9: heated_plate_openmp 的内存消耗

3 实验感想

- 本次实验第一次使用了 **valgrind massif** 工具采集内存消耗，这是之前在编写程序过程中没有去注意到的方面。程序的性能评价不止时间复杂度（对应运算时间），还有空间复杂度（对应内存消耗）。学习使用 valgrind massif 可以帮助程序员更好地优化代码，找到错误以提高程序性能。
- 本次实验主要比较了 MPI 与 OpenMP 的并行效果，OpenMP 所有线程共享内存空间，硬件制约较大，这也帮助它有效地针对循环并行化，但要使用 critical 等处理冲突。MPI 线程有自己的内存和变量，这样不用担心冲突问题，但性能上会受到通信网络的影响，内存消耗也大。
- 本次实验由于之前实验留下的隐患，导致 parallel_for 函数的并行计算效果较差，不仅没有充分利用程序的局部性原理，而出现了内存泄漏的问题还找不到源头，以后编写代码需要引以为戒。