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 计科 8 班	专业（方向）	超算
学号	18340208	姓名	张洪宾
Email	2285075600@qq.com	完成时间	2020 年 12 月 2 号

目录

1 实验目的

- 通过解决实际问题，来更加深入地理解并行计算的应用。
- 通过用不同的方式对某个问题进行优化，从而更加理解高性能计算的意义。
- 了解并使用 valgrind 工具来对内存进行分析。

2 实验过程及核心代码

2.1 实验环境

因为这次的问题比较复杂，并且在第二问的时候需要开多个进程，电脑上的虚拟机已经很难满足运行所需，所以我在这里使用了超算的 th2k 集群来作为实验环境。

2.1.1 硬件

- Intel(R) Xeon(R) Gold 6150 CPU @ 2.70GHz

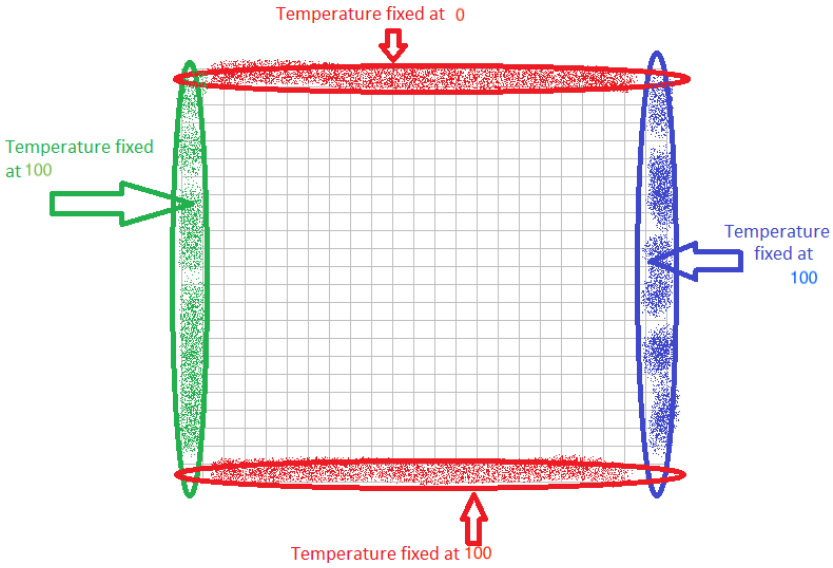
2.1.2 软件

- 操作系统：CentOS Linux release 7.6.1810 (Core)
- Intel Parallel Studio XE 2018
- gcc version 4.8.5
- valgrind-3.13.0
- slurm 资源管理系统

2.2 问题分析

我们要解决的问题是一个热学问题：求解二维稳态热方程。在这里我们要求解在一个矩形四条边缘的温度已知的情况下，内部各点的温度情况。

问题如下：



如图，我们已知矩形四条边缘的温度，我们需要求的是该矩形中每个点的温度，对应的就是图中每个格子的温度情况。

根据维基百科的解释，我们大致是需要找到一个平衡态，在这个状态下每个点的温度不会在发送变化。转换成算法的话，则是先让中间的所有方格的温度都取四个边缘的温度的平均值，然后不断的做热传导的模拟，直到对于每个点，每次热传导带来的温度都小于一个阈值，在我们这个程序取为0.001，在这个时候就达到了热平衡状态了。

定义该矩形为二维数组 $w[M][N]$，再定义 $u[M][N]$ 为矩形的备份，每次热传导的模拟可以用下面的公式表示：

$$w[i][j] = \frac{1}{4} \times (u[i-1][j] + u[i][j-1] + u[i+1][j] + u[i][j+1])$$

即每次热传导时当前方格的温度等于其四周的方格的温度的均值。这也符合我们的主观判断。

了解了具体的问题和其解决方法后我们就可以开始来改进原有的代码了。

2.3 用自己实现的 parallel_for 库来替代 omp for

2.3.1 代码的修改

在上个实验中我们实现了 parallel_for 库，在这里只要针对每一个 omp for 实现一个简单的 factor，就可以调用 parallel_for 库中的 parallel_for 函数来自己实现并行了。

上次我封装的库函数如下：

```
1 void parallel_for(int start,int end,int increment,void *(*functor)(void*),void
    *arg,int num_threads,int chunksize){
```

```
    struct parallel_arg argument[num_threads];
    for(int i = start;i < end;i += increment){
        argument[((i - start) / (chunksize * increment) ) % num_threads].index.
            push_back(i);//将索引按块划分给指定线程
    }
    pthread_t tid[num_threads];
    for(int i = 0;i < num_threads;i += increment){
        argument[i].fun_arg = arg;
        argument[i].increment = increment;
        pthread_create(&tid[i],NULL,functor,&argument[i]);
    }
    for(int i = 0;i < num_threads;i++){
        pthread_join(tid[i],NULL);
    }
}
```

对于每一个 omp for，我们都可以利用针对这个函数，写出一个 factor，进而替代原来的 omp for。在这里以核心的循环————每次迭代计算热传导后新的温度的循环为例，将其改造为如下的 factor:

```
void * iteration(void* arg){
    parallel_arg* parallel_arg = reinterpret_cast<struct parallel_arg*> (arg);
    std::vector<int> index = parallel_arg->index;
    int increment = parallel_arg->increment;
    struct arrays*temp = reinterpret_cast<struct arrays*>(parallel_arg->fun_arg
        );
    double * w = temp->w;
    double * u = temp->u;

    for(int i = 0;i < index.size();i += increment){
        #pragma ivdep
        for(int j = 1;j < N - 1;j++){
            w[index[i] * N + j] = (u[(index[i] - 1) * N + j] + u[(index[i] + 1)
                * N + j] + u[index[i] * N + j - 1] + u[index[i] * N + j + 1]) /
                4.0;
        }
    }
}
```

观察可以得到，我们首先根据 parallel_for 函数中传来的 buf，获取具体的数据，然后根据循环的范围以及线程的数量对循环进行划分。这也是设计其他 factor 的基本方法。

为了传递 u 和 w 的地址，我定义了如下结构体：

```
struct arrays{
    double * w;
    double * u;
};
```

然后我们就可以在 main 函数中将 w 和 u 的地址存入其中，然后用如下的代码调用 parallel_for 循环：

```
    parallel_for(1,M - 1,1,iteration,&w_and_u,thread_count,1);
```

其他部分的 omp for 的替代也是类似的方法，在此就不再赘述。

2.3.2 调用 Intel 的编译器来优化运行

在这里我继续参考了上次使用的 Intel 教程:英特尔 ® 至强融核 ™ 处理器优化教程。与实验 4 类似，我针对 Intel 的 Xeon 处理器做了一些优化。

在这里我的编译运行脚本 run.sh 如下：

```
icpc -fPIC -shared -o libparallel_for.so parallel_for.cpp -march=native -O3 -Ofast -no-prec-div -fp-model fast=2 -par-affinity=compact -Iinclude/
icpc heated_plate_openmp.cpp -L. -lparallel_for -pthread -march=native -fopenmp -O3 -Ofast -no-prec-div -fp-model fast=2 -par-affinity=compact
export LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:."
export OMP_NUM_THREADS=36
numactl -m 0 ./a.out
```

具体的优化的解释在上次的实验报告中已经提过了，在此不再赘述。

2.3.3 编写 slurm 脚本来进行调度

因为在集群上有很多的任务在运行，如果直接 salloc 节点的话可能要等待很久。因此我选择了编写 slurm 脚本 parallel_for.slurm 并提交上去，然后直接等待输出的 slurm-id.out 文件就行了。

slurm 脚本中指定了节点的分区，还有节点的数量，每个节点包含的进程，任务的名称等等，具体如下：

```
#!/bin/bash
#SBATCH -J zhb -p work
#SBATCH -N 1
```

```
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=36
module load intel/18.0.1
module load IMPI/2018.1.163-icc-18.0.1
module load opt/numactl/2.0.13
sh run.sh
```

在运行 run.sh 之前，先用 module load 加载了需要的 Intel 编译器。然后就可以运行程序了。

我们可以用 sbatch parallel_for.slurm 提交运行脚本，然后用 squeue 查看任务队列，结果如下：

```
[sysu_hpcedu_302@ln102 ~/asc21/zhb/lab5/Q1]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
            637080      work      hxt sysu_hpc  R    2:09:30      1 cpn213
            637108      work       lj sysu_hpc  R    1:53:11      1 cpn249
            637037      work     bash sysu_hpc  R    3:56:15      1 cpn106
            637147      work      zhb sysu_hpc  R       0:26      1 cpn222
            635368  gpu_v100      fjx sysu_hpc  R 1-08:07:55      2 gpu[29,51]
```

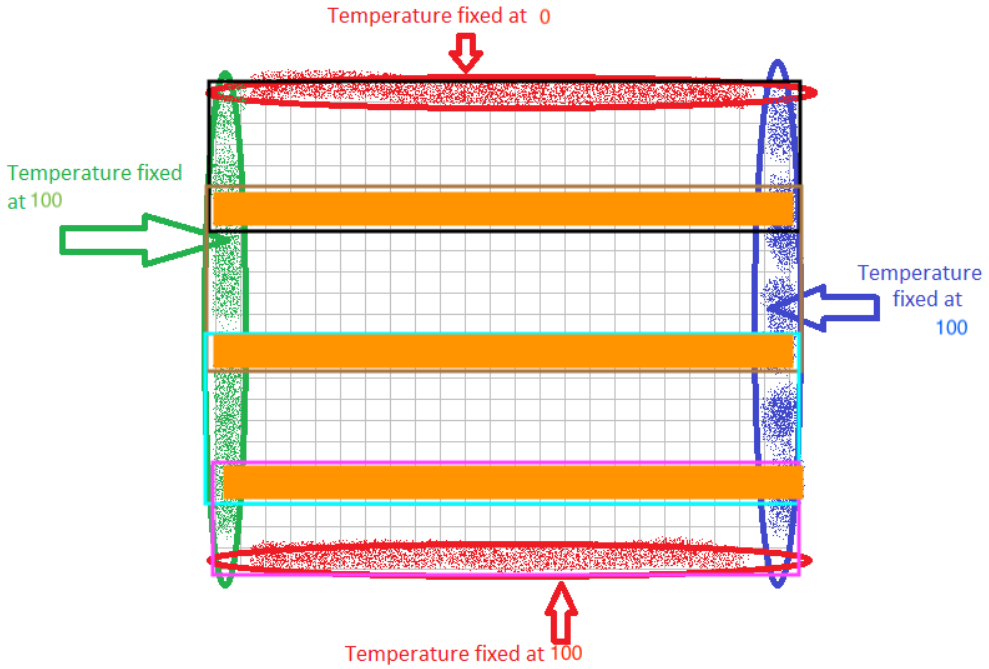
任务执行完毕后，会将输出的内容保存到 slurm-JOBID.out 文件中，可以查看这些文件来查看运行的结果。

2.4 将 heated_plate_openmp 改造成 MPI 应用

2.4.1 代码的修改

在这里的修改比较复杂，因为在共享内存的编程模型如 pthread 和 openMP 中，我们不需要去考虑进程之间的通信，这样子就不会涉及这方面 Send 和 Recv 的代码。但是在 MPI 中，由于内存不是共享的，所以我们需要做信息的通信。

在这里我将问题做如下划分：



如图，在原问题上，每个方框就代表一个进程的计算范围，因为每次热传导过程中，无法对边缘进行更新，所以我们需要适当地增大进程计算范围，使得在一次热传导过程中，所有进程的计算可以更新所有的方格。

然后图中橙色的部分就是所谓的边缘，在这里每个橙色的部分各占两行，两个相邻的进程都拥有其中的一部分。在每次计算完毕后，任意相邻两个进程都计算了橙色部分中靠近自己的一行，而另外一行没有计算，这时候我们就可以用 MPI_Pack 将这些边缘封装起来，然后用 MPI_Send 发送给相邻的进程。这样子就可以完成对整个边缘的更新。

然后每次计算迭代后的变化量时，我们只需要计算每个进程管辖的行的范围的最大误差，然后再用 MPI_Allreduce 来获取所有进程的最大误差，当误差小于给定的阈值的时候，结束 while 循环，这一部分也与 openMP 的程序中的结构相一致。

完成了以上的修改，最核心的代码如下：

```
while ( epsilon <= diff )
{

    memcpy(&u[0][0],&w[0][0],sizeof(double) * M * N);


    for ( i = max(1,my_id * M / size); i < min(M - 1,(my_id + 1) * M / size); i
        ++ )
    {
```

```
9           for ( j = 1; j < N - 1; j++ )
10          {
11          w[i][j] = ( u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] ) / 4.0;
12          }
13      }
14      diff = 0.0;
15      double buf1[N],buf2[N];
16      int pos1 = 0,pos2 = 0;
17      if(my_id != size - 1){
18          MPI_Pack(&w[(my_id + 1) * M / size - 1][1],N - 2,MPI_DOUBLE,buf1,8 * (N
                - 2),&pos1,MPI_COMM_WORLD);
19      }
20      if(my_id == 0){
21          MPI_Send(buf1,N - 2,MPI_DOUBLE,my_id + 1,0,MPI_COMM_WORLD);
22      }
23      else if(my_id % 2 == 0 && my_id != size - 1){
24          MPI_Send(buf1,N - 2,MPI_DOUBLE,my_id + 1,0,MPI_COMM_WORLD);
25          MPI_Recv(buf2,N - 2,MPI_DOUBLE,my_id - 1,0,MPI_COMM_WORLD,&status);
26      }
27      else if(my_id % 2 && my_id != size - 1){
28          MPI_Recv(buf2,N - 2,MPI_DOUBLE,my_id - 1,0,MPI_COMM_WORLD,&status);
29          MPI_Send(buf1,N - 2,MPI_DOUBLE,my_id + 1,0,MPI_COMM_WORLD);
30      }
31      else if(my_id == size - 1){
32          MPI_Recv(buf2,N - 2,MPI_DOUBLE,my_id - 1,0,MPI_COMM_WORLD,&status);
33      }
34      //&w[my_id * M / size - 1][1]
35      if(my_id != 0){
36          MPI_Unpack(buf2,8*(N - 2),&pos2,&w[my_id * M / size - 1][1],N - 2,
                MPI_DOUBLE,MPI_COMM_WORLD);
37      }
38      pos1 = pos2 = 0;
39      if(my_id != 0){
40          MPI_Pack(&w[my_id * M / size][1],N - 2,MPI_DOUBLE,buf1,8 * (N - 2),&
                pos1,MPI_COMM_WORLD);
41      }
42      if(my_id == 0){
```

```
43          MPI_Recv(buf2,N - 2,MPI_DOUBLE,my_id + 1,0,MPI_COMM_WORLD,&status);
44      }
45      else if(my_id % 2 == 0 && my_id != size - 1){
46          MPI_Recv(buf2,N - 2,MPI_DOUBLE,my_id + 1,0,MPI_COMM_WORLD,&status);
47          MPI_Send(buf1,N - 2,MPI_DOUBLE,my_id - 1,0,MPI_COMM_WORLD);
48      }
49      else if(my_id % 2 && my_id != size - 1){
50          MPI_Send(buf1,N - 2,MPI_DOUBLE,my_id - 1,0,MPI_COMM_WORLD);
51          MPI_Recv(buf2,N - 2,MPI_DOUBLE,my_id + 1,0,MPI_COMM_WORLD,&status);
52      }
53      else if(my_id == size - 1){
54          MPI_Send(buf1,N - 2,MPI_DOUBLE,my_id - 1,0,MPI_COMM_WORLD);
55      }
56
57      if(my_id != size - 1){
58          MPI_Unpack(buf2,8*(N - 2),&pos2,&w[(my_id + 1) * M / size][1],N - 2,
                MPI_DOUBLE,MPI_COMM_WORLD);
59      }
60
61      my_diff = 0.0;
62
63
64      for ( i = max(1,my_id * M / size); i < min(M - 1,(my_id + 1) * M / size); i
           ++ )
65      {
66          for ( j = 1; j < N - 1; j++ )
67          {
68              if ( my_diff < fabs ( w[i][j] - u[i][j] ) )
69              {
70              my_diff = fabs ( w[i][j] - u[i][j] );
71              }
72          }
73      }
74
75      MPI_Allreduce(&my_diff,&diff,1,MPI_DOUBLE,MPI_MAX,MPI_COMM_WORLD);
76  }
```

这里的通信比较复杂，因此代码量较大，同时使用了 MPI_Pack 简化了发送的过程。

2.4.2 编写 slurm 脚本来进行调度

同样的，根据这个问题，我们也可以编写一个 slurm 脚本 MPI.slurm 来进行调度，如下：

```
#!/bin/bash
#SBATCH -J zhb -p work
#SBATCH -N 8                          # 申请 8 个节点
#SBATCH --ntasks-per-node=1           # 每个节点开 1 个进程
#SBATCH --cpus-per-task=1
module load intel/18.0.1
module load IMPI/2018.1.163-icc-18.0.1
export I_MPI_FAVRICS=shm:dapl
mpirun ./a.out
```

用 sbatch MPI.slurm 提交任务，等待之后就可以得到结果了。

2.5 性能分析实验

2.5.1 通过不同的线程/进程来比较二者的性能

在这里我首先运行原始的代码，然后测量在不同规模下的运行时间，然后再在不同规模和不同线程/进程下运行，生成的 slurm 脚本改名为“问题规模 _ 并发度”，如 2000_4。

2.5.2 用 valgrind massif 工具采集内存消耗情况进行分析

在这里由于集群上没有安装 valgrind massif，而且我也没有 root 权限，我只能在自己的文件夹下下载源码进行安装。

首先下载 valgrind-3.16.1.tar.bz2，然后用 tar -xjvf valgrind-3.16.1.tar.bz2 解压，用 module load 加载 autoconf 工具后，用./autogen.sh 来设置环境。设置好后，用./configure –prefix= 我的路径来设置安装目录，然后就使用 make && make install 来安装即可。

在使用之前，需要用如下命令将 valgrind 加入环境变量：

```
export PATH="${PATH}:/GPUFS/sysu_hpcedu_302/asc21/zhb/autoconf/bin"
export PATH="${PATH}:/GPUFS/sysu_hpcedu_302/asc21/zhb/automake/bin"
export PATH="${PATH}:/GPUFS/sysu_hpcedu_302/asc21/zhb/valgrind/bin"
```

其中的路径是我的安装目录。

然后就可以使用 valgrinde 了：

3 实验结果

3.1 在不同线程下原代码的运行时间

在集群上用 salloc -N 1 申请一个节点,然后用编译脚本编译,并用 export OMP_NUM_THREADS=n 来指明是 n 个线程。

运行的结果如下:

线程数量 \ 问题规模	250	500	1000	2000
1	6.339595	43.163215	184.181825	755.944196
2	3.271074	21.818635	92.659870	375.049256
4	1.703748	11.103726	46.833873	203.216079
8	0.975737	5.865594	23.977993	153.701082

表 1: 不同问题规模不同线程下的运算时间 (s)

不难看出随着问题的增大，花费的时间会变得非常巨大。而且线程数量增加会带来一定的加速。

3.2 用 parallel_for 代替了 omp for 之后的运行时间

由于我自己实现的时候没有考虑一个线程或一个进程的时候会出现什么情况，且在这两种情况下并没有使用 parallel_for 或 MPI 的意义，因此我只测量了在线程/进程数量为 2、4、8 的运行时间。

由于之前使用了 slurm 脚本运行，所以可以直接用 cat 查看 slurm 生成的 out 文件，在这里我将它们改名放在 src/Q1/res 文件夹下，名字又问题规模和线程数量组成。

统计的结果如下:

线程数量 \ 问题规模	250	500	1000	2000
2	1.376457	10.441163	42.210864	307.264974
4	1.909977	12.536731	51.421372	339.307900
8	2.926036	13.845252	50.716549	335.346331

表 2: 不同问题规模不同线程下的运算时间 (s)

在这里我发现问题规模比较小的时候，时间很短，问题规模较大的时候，运行时间会变得非常长，说明这个代码的可扩展性比较差。

3.3 实现的 MPI 版本的运行时间

这部分的 slurm 输出文件在 Q2/res 文件夹下，结果如下：

线程数量 \ 问题规模	250	500	1000	2000
2	8.505036	8.378345	39.467277	179.416358
4	6.385561	6.439662	33.055873	133.864270
8	5.441996	5.446275	29.362248	120.626342

表 3: 不同问题规模不同进程下的运算时间 (s)

从这里可以看出，程序在问题规模较小的时候，运行时间较长，随着问题规模的增大，时间增长的不是很明显。这是因为在问题规模比较小的时候，通信的开销占了比较大的比重，而问题规模较大的时候，这部分的开销所占用的比重就减小了，所以相比其他两种运行方式的情况开销更小。

3.4 用 valgrind 工具分析内存的情况

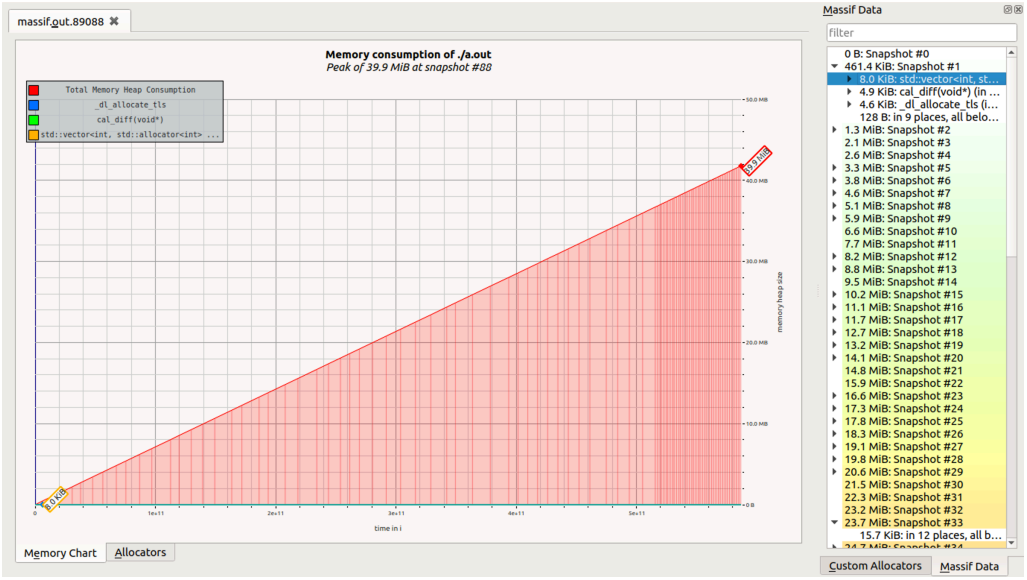
在这里将 valgrind 加入环境变量后，然后用如下命令执行：

```
valgrind --tool=massif --stacks=yes [my commands]
```

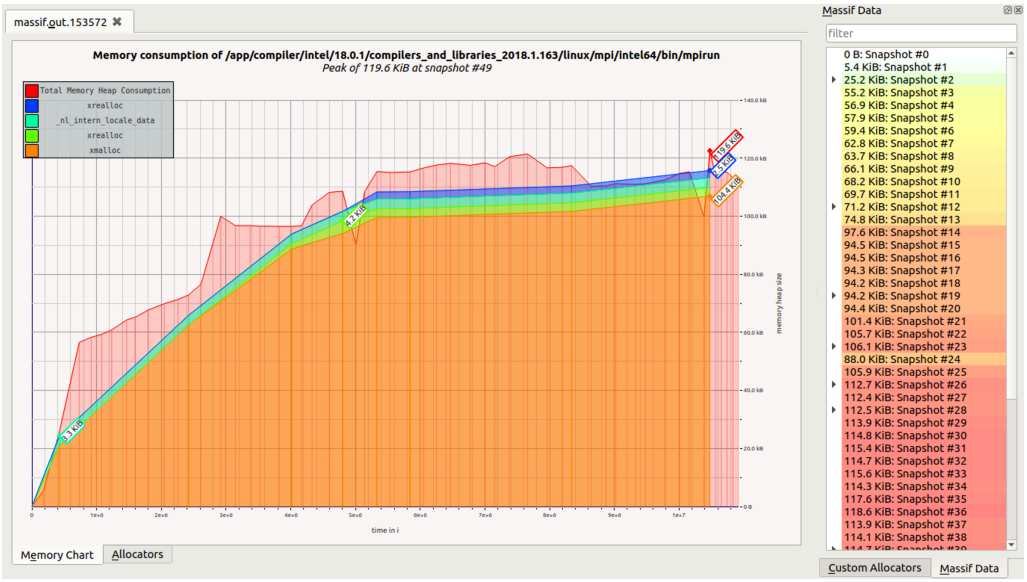
所以我们在 parallel_for 的问题中用 valgrind –tool=massif –stacks=yes ./a.out 来生成结果，然后在 MPI 版本中用 valgrind –tool=massif –stacks=yes mpirun ./a.out 来生成结果。

在这里我在本地安装了 massif visualizer，用可视化界面来代替 ms_print 来更好地展示。在这里只生成了问题规模为 2000 时 parallel_for 版本和 MPI 版本运行后的 massif.out 文件，用 massif visualizer 查看，结果如下：

parallel_for 版本下，问题规模为 2000 的时候运行的结果：



MPI 版本下，问题规模为 2000 的时候的运行结果：



值得提到的是，MPI 生成了多个文件，不过每个文件都大同小异，所以我就取其中一个。

通过分析 massif.out 文件，两种情况下，占用内存较多的都是内存的申请，我在代码中没有显式地申请内存，猜测是 pthread 和 MPI 底层在实现的时候进行了申请。

可以看出，在问题规模很大的时候，parallel_for 版本在堆上申请的内存远远大于 MPI 版本，这可能会造成诸如大量的 cache-miss 之类的问题，导致性能严重下降。这可能是因为我具体实现的库函数没有考虑到底层的细节问题，在这方面的优化远远不如迭代了多个版本的 openMP 和 MPI 库。

4 实验感想

- 集群的单核性能也不是很强，但是胜在核心数量，我自己测试了一下并行度为 32 和 64 的时候的运行情况，性能更优。
- 采用 Intel 编译器针对具体架构优化性能提升的较明显。
- Linux 下采用源码编译软件再配置环境变量的方式会防止出现依赖问题，更加可取。
- 内存在很大程度上容易成为程序运行的瓶颈，当内存分配不当的时候可能会导致性能的下降。