Lab 5: fft 和 headted plate 的并行化

学号：19335109	课程：高性能计算
姓名：李雪堃	学期：Fall 2021
专业：计算机科学与技术（超算）	教师：黄聃、卢宇彤
邮箱：i@xkun.me	TAs：江嘉治、刘亚辉

Table of Contents

(一) 实验任务

- 任务 1：
  - 通过实验 4 构造的基于 Pthreads 的 `parallel_for` 函数替换 fft_serial 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。
- 任务 2：
  - 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）。
  - 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 和并行化 fft 应用，包括：
    - 不同问题规模的并行化 fft 应用并行执行时间对比，其中问题规模定义为 N 变化范围 2， 4， 6， 8， 16， 32，64，128，……， 2097152；并行规模为 1，2，4，8 进程/线程。

- 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。

(二) 实验环境

- Ubuntu 20.04.3 LTS x86_64
- gcc 9.3.0
- OpenMPI 4.0.3
- GNU Make 4.2.1
- valgrind 3.15.0

(三) 实验过程和核心代码

(1) parallel_for 并行化 fft

代码在 `fft/fft-pf` 目录下。其中 `parallel_for.h` 和 `parallel_for.c` 是上次实验基于 pthreads 的 `parallel_for` 函数。上次写的有些 bug，这次进行了修复和改进。

```
typedef struct
{
  int my_rank;  // these four arguments are specified by parallel_for
  int my_start;
  int my_end;
  int my_increment;

  void * func_arg;  // thread function arguments, specified by user
} pf_arg_t;
```

```
void parallel_for(int start, int end, int increment, \
                  void *(*functor)(void *), void *arg, int thread_count);
```

首先是 `parallel_for` 的内部参数 `pf_arg_t`，这个参数不需要用户指定，是在 `parallel_for` 函数内部传递给各个线程的参数。 `pf_arg_t` 中的 `fun_arg` 才是用户自定义的结构体参数， `parallel_for` 中的 `void *arg` 参数传递的就是用户自定义的结构体。

```c
void parallel_for(int start, int end, int increment, \
                  void *(*functor)(void *), void *func_arg, int thread_count)
// functor is a function pointer, defined by user, executed by thread
// func_arg is the arguments of functor, defined and allocated by user
{
  pthread_t *thread_handles = (pthread_t *) malloc(sizeof(pthread_t) * thread_count);
  pf_arg_t *pf_args = (pf_arg_t *) malloc(sizeof(pf_arg_t) * thread_count);

  int loop_count; // total loop times
  if ((end - start) / increment == 0)
    loop_count = (end - start) / increment;
  else
    loop_count = (end - start) / increment + 1;

  if (loop_count <= thread_count) // if loop_count smaller than thread_count
  {
    thread_count = 1;
  }

  int my_count = loop_count / thread_count;   // expected average thread loop times

  for (long thread = 0; thread < thread_count; thread++)
  {
    if (thread == thread_count - 1) // I'm the last thread
    {
      pf_args[thread].my_rank = thread;
      pf_args[thread].my_start = start + increment * my_count * thread;
      pf_args[thread].my_end = end;
      pf_args[thread].my_increment = increment;
      pf_args[thread].func_arg = func_arg;
    }
    else  // I'm not the last thread
    {
      pf_args[thread].my_rank = thread;
      pf_args[thread].my_start = start + increment * my_count * thread;
      pf_args[thread].my_end = pf_args[thread].my_start + increment * my_count;
      pf_args[thread].my_increment = increment;
      pf_args[thread].func_arg = func_arg;
    }
  }

  for (long thread = 0; thread < thread_count; thread++)
    pthread_create(&thread_handles[thread], NULL, functor, (void *)&pf_args[thread]);

  for (long thread = 0; thread < thread_count; thread++)
    pthread_join(thread_handles[thread], NULL);

  free(thread_handles);
  free(pf_args);
}
```

在 `parallel_for` 函数中，首先计算总的循环次数 `loop_count`，然后判断 `loop_count` 是否小于等于线程数 `thread_count`，如果是则只开启一个线程，相当于串行计算，这样做是为了不损失性能。

接着是初始化线程函数的参数，将 `pf_arg_t` 中的 `func_arg` 指针指向用户传递来的参数结构体指针。最后是线程创建和销毁。

在 fft_serial 中，经过观察分析，发现 `cffti` 和 `step` 函数可以并行，并且测试后的确带来了性能的提升；`ccopy` 也可以并行，但并行后性能反而下降。

首先是 `cffti`，它的作用是准备 FFT 计算时需要的 sin 和 cos 表。首先创建参数结构体 `cffti_arg_t`，然后编写 `cffti_pf` 函数，从 `pf_arg_t` 中获得 `my_start`、`my_end` 和 `my_increment`，以及用户传递的参数结构体。然后获取这些参数，用一个循环进行计算即可。

```c
typedef struct
{
  double aw;
  double *w;
} cffti_arg_t;
```

```c
void *cffti_pf(void *arg)
{
  pf_arg_t * my_arg = (pf_arg_t *)arg;
  int my_start = my_arg->my_start;
  int my_end = my_arg->my_end;
  int my_increment = my_arg->my_increment;

  cffti_arg_t * cffti_arg = (cffti_arg_t *)my_arg->func_arg;
  double aw = cffti_arg->aw;
  double *w = cffti_arg->w;
  double x;

  // #ifdef DEBUG
  //   int my_rank = my_arg->my_rank;
  //   printf("I'm thread %d, my_start = %d, my_end = %d\n", my_rank, my_start, my_end);
  // #endif

  for (int i = my_start; i < my_end; i += my_increment)
  {
    x = aw * ((double)i);
    w[i * 2 + 0] = cos(x);
    w[i * 2 + 1] = sin(x);
  }

  return NULL;
}
```

在 `cffti` 函数中，将原有的循环替换为 `parallel_for` 即可。

```c
void cffti(int n, double w[])
/*
  Purpose:
    CFFTI sets up sine and cosine tables needed for the FFT calculation.
  Parameters:
    Input, int N, the size of the array to be transformed.
    Output, double W[N], a table of sines and cosines.
*/
{
  double aw;
  int n2;
  const double pi = 3.141592653589793;

  n2 = n / 2;
  aw = 2.0 * pi / ((double)n); // aw = 2pi / n

  cffti_arg_t cffti_arg;
  cffti_arg.aw = aw;
  cffti_arg.w = w;

  parallel_for(0, n2, 1, cffti_pf, (void *)&cffti_arg, thread_count);
}
```

然后是 `step` 函数的并行，该函数作用是执行 FFT 中的一次迭代。同样地，我们需要创建 `step_arg_t` 参数结构体。然后编写 `step_pf` 函数，解析参数，执行循环。

```c
typedef struct
{
  int mj;
  int mj2;
  double *a;
  double *b;
  double *c;
  double *d;
  double *w;
  double sgn;
} step_arg_t;
```

```c
void *step_pf(void *arg)
{
  pf_arg_t *my_arg = (pf_arg_t *)arg;
  int my_start = my_arg->my_start;
  int my_end = my_arg->my_end;
  int my_increment = my_arg->my_increment;

  step_arg_t *step_arg = (step_arg_t *)my_arg->func_arg;
  int mj = step_arg->mj;
  int mj2 = step_arg->mj2;
  double *a = step_arg->a;
  double *b = step_arg->b;
  double *c = step_arg->c;
  double *d = step_arg->d;
  double *w = step_arg->w;
  int sgn = step_arg->sgn;

  double ambr;
  double ambu;
  int j;
  int ja;
  int jb;
  int jc;
  int jd;
  int jw;
  int k;
  double wjw[2];

  for (j = my_start; j < my_end; j += my_increment)
  {
    jw = j * mj;
    ja = jw;
    jb = ja;
    jc = j * mj2;
    jd = jc;

    wjw[0] = w[jw * 2 + 0];
    wjw[1] = w[jw * 2 + 1];

    if (sgn < 0.0)
    {
      wjw[1] = -wjw[1];
    }

    for (k = 0; k < mj; k++)
    {
      c[(jc + k) * 2 + 0] = a[(ja + k) * 2 + 0] + b[(jb + k) * 2 + 0];
      c[(jc + k) * 2 + 1] = a[(ja + k) * 2 + 1] + b[(jb + k) * 2 + 1];

      ambr = a[(ja + k) * 2 + 0] - b[(jb + k) * 2 + 0];
      ambu = a[(ja + k) * 2 + 1] - b[(jb + k) * 2 + 1];

      d[(jd + k) * 2 + 0] = wjw[0] * ambr - wjw[1] * ambu;
      d[(jd + k) * 2 + 1] = wjw[1] * ambr + wjw[0] * ambu;
    }
  }
  return NULL;
}
```

`step` 函数在 `cfft2` 函数中被调用，我们将其中的 `step` 函数全部替换为 `parallel_for` 函数，开启线程执行 `step_pf`。

```c
void cfft2(int n, double x[], double y[], double w[], double sgn)
/*
  Purpose:
    CFFT2 performs a complex Fast Fourier Transform.
  Parameters:
    Input, int N, the size of the array to be transformed.
    Input/output, double X[2*N], the data to be transformed.
    On output, the contents of X have been overwritten by work information.
    Input, double W[N], a table of sines and cosines.
    Input, double SGN, is +1 for a "forward" FFT and -1 for a "backward" FFT.
    Output, double Y[2*N], the forward or backward FFT of X.
*/
{
  int j;
  int m;
  int mj;
  int mj2;
  int lj;
  int tgle;
  step_arg_t step_arg;

  m = (int)(log((double)n) / log(1.99)); // m = log_2^n
  mj = 1;
  mj2 = 2 * mj;
  lj = n / mj2;

  tgle = 1; // Toggling switch for work array.
  step_arg.mj = mj;
  step_arg.mj2 = mj2;
  step_arg.a = &x[0 * 2 + 0];
  step_arg.b = &x[(n / 2) * 2 + 0];
  step_arg.c = &y[0 * 2 + 0];
  step_arg.d = &y[mj * 2 + 0];
  step_arg.w = w;
  step_arg.sgn = sgn;
  parallel_for(0, lj, 1, step_pf, (void *)&step_arg, thread_count);
  // step(n, mj, &x[0 * 2 + 0], &x[(n / 2) * 2 + 0], &y[0 * 2 + 0], &y[mj * 2 + 0], w, sgn);

  if (n == 2)
  {
    return;
  }

  for (j = 0; j < m - 2; j++) // m = log_2^n, 这里的 step 要被重复执行很多次，线程开销过大
  {
    mj = mj * 2;
    mj2 = 2 * mj;
    lj = n / mj2;
    if (tgle)
    {
      step_arg.mj = mj;
      step_arg.mj2 = mj2;
      step_arg.a = &y[0 * 2 + 0];
      step_arg.b = &y[(n / 2) * 2 + 0];
      step_arg.c = &x[0 * 2 + 0];
      step_arg.d = &x[mj * 2 + 0];
      step_arg.w = w;
      step_arg.sgn = sgn;

      parallel_for(0, lj, 1, step_pf, (void *)&step_arg, thread_count);
      // step(n, mj, &y[0 * 2 + 0], &y[(n / 2) * 2 + 0], &x[0 * 2 + 0], &x[mj * 2 + 0], w, sgn);
      tgle = 0;
    }
    else
    {
      step_arg.mj = mj;
      step_arg.mj2 = mj2;
      step_arg.a = &x[0 * 2 + 0];
      step_arg.b = &x[(n / 2) * 2 + 0];
      step_arg.c = &y[0 * 2 + 0];
      step_arg.d = &y[mj * 2 + 0];
      step_arg.w = w;
      step_arg.sgn = sgn;

      parallel_for(0, lj, 1, step_pf, (void *)&step_arg, thread_count);
      // step(n, mj, &x[0 * 2 + 0], &x[(n / 2) * 2 + 0], &y[0 * 2 + 0], &y[mj * 2 + 0], w, sgn);
      tgle = 1;
    }
  }
```

```
80
81    if (tgle) // Last pass through data: move Y to X if needed.
82    {
83      ccopy(n, y, x);
84    }
85
86    mj = n / 2;
87    mj2 = mj * 2;
88    lj = n / mj2;
89    step_arg.mj = mj;
90    step_arg.mj2 = mj2;
91    step_arg.a = &x[0 * 2 + 0];
92    step_arg.b = &x[(n / 2) * 2 + 0];
93    step_arg.c = &y[0 * 2 + 0];
94    step_arg.d = &y[mj * 2 + 0];
95    step_arg.w = w;
96    step_arg.sgn = sgn;
97    parallel_for(0, lj, 1, step_pf, (void *)&step_arg, thread_count);
98    // step(n, mj, &x[0 * 2 + 0], &x[(n / 2) * 2 + 0], &y[0 * 2 + 0], &y[mj * 2 + 0], w, sgn);
99
100   return;
101 }
```

下面是运行结果。

fft_pf_test.txt M

Lab > lab5 > fft > asset > fft_pf_test.txt

```
4    C/PARALLEL-FOR version
5
6    Demonstrate an implementation of the Fast Fourier Transform
7    of a complex data vector.
8
9    Number of threads =             12
10
11   Accuracy check:
12
13     FFT ( FFT ( X(1:N) ) ) == N * X(1:N)
14
15               N      NITS    Error         Time          Time/Call     MFLOPS
16
17               2     10000  7.859082e-17  2.421576e-01  1.210788e-05      0.825908
18               4     10000  1.209837e-16  4.935291e-01  2.467646e-05      1.620978
19               8     10000  6.820795e-17  7.275663e-01  3.637832e-05      3.298668
20              16     10000  1.438671e-16  1.002143e+00  5.010717e-05      6.386311
21              32      1000  1.331210e-16  3.922027e-01  1.961013e-04      4.079524
22              64      1000  1.776545e-16  6.525185e-01  3.262592e-04      5.884891
23             128      1000  1.929043e-16  9.471074e-01  4.735537e-04      9.460384
24             256      1000  2.092319e-16  1.191531e+00  5.957656e-04     17.187968
25             512       100  1.927488e-16  1.439251e-01  7.196256e-04     32.016650
26            1024       100  2.312093e-16  1.904348e-01  9.521740e-04     53.771687
27            2048       100  2.445006e-16  2.037586e-01  1.018793e-03    110.562214
28            4096       100  2.476589e-16  2.468777e-01  1.234389e-03    199.094509
29            8192        10  2.571250e-16  2.745175e-02  1.372587e-03    387.938866
30           16384        10  2.736298e-16  3.632549e-02  1.816274e-03    631.446473
31           32768        10  2.924127e-16  4.803912e-02  2.401956e-03   1023.166141
32           65536        10  2.833553e-16  7.481372e-02  3.740686e-03   1401.582584
33          131072         1  3.142312e-16  1.202956e-02  6.014778e-03   1852.290986
34          262144         1  3.216005e-16  2.524719e-02  1.262360e-02   1868.957224
35          524288         1  3.282664e-16  5.299924e-02  2.649962e-02   1879.549865
36         1048576         1  3.284479e-16  1.229341e-01  6.146704e-02   1705.915969
37         2097152         1  3.509548e-16  2.546743e-01  1.273371e-01   1729.275146
38
39   FFT_PARALLEL_FOR:
40     Normal end of execution.
41
42   03 December 2021 09:51:36 PM
43
```

(2) heated_plate_openmp 改造为基于 MPI 的进程并行应用

heated plate 是个数值计算问题，主要是通过迭代来计算 plate 上温度收敛状态的分布。

一个 plate 用一个 MxN 的矩形区域来表示，程序中设置 M = N = 500，开始时每个节点都赋予一个初始的温度，程序中设置左边界、右边界和下边界的温度都为 100，上边界的温度为 0，中间位置的温度是边界温度的均值（所有边界节点温度的和除以边界节点数）。这个过程是初始化的过程，不好并行，我因此用 master 进程来初始化，在 master 进程中使用 openmp 并行。

迭代的过程非常简单，对面中间位置的节点（不在边界的节点），每轮更新它的温度为其相邻节点温度的均值（上下左右四个节点），即：

$$w[center] = \frac{w[north] + w[south] + w[east] + w[west]}{4}, \ for\ each\ iteration$$

一直进行下去，直到相对误差（上次迭代结果与本次迭代结果相比）小于一个阈值，误差的计算是采取所有节点中误差最大值作为本次迭代的误差。

首先是 master 进程（rank 0）初始化 plate。内部用 openmp 并行加快速度。比较简单，这里不再赘述。

```
if (my_rank == 0) // I'm the master
  {
    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 processes = %d\n", comm_sz);

    int i, j;
    #pragma omp parallel shared(w) private(i, j)
    {
      // 初始化 w 边界的值
      #pragma omp for
      for (i = 1; i < M - 1; i++)
      {
        w[i][0] = 100.0;      // left border
        w[i][N - 1] = 100.0;  // right border
      }
      #pragma omp for
      for (j = 0; j < N; j++)
      {
        w[M - 1][j] = 100.0;  // bottom border
        w[0][j] = 0.0;        // top border
      }

      // 计算边界 w 值之和，加到 mean 上
      #pragma omp for reduction(+ : mean)
      for (i = 1; i < M - 1; i++)
      {
        mean += w[i][0] + w[i][N - 1];
      }
      #pragma omp for reduction(+ : mean)
      for (j = 0; j < N; j++)
      {
        mean += w[M - 1][j] + w[0][j];
      }
    }

    // 算均值，除以边界节点的个数
    mean = mean / (double)(2 * M + 2 * N - 4);
    printf("\n");
    printf("  MEAN = %f\n", mean);

    // 初始化 w 内部为 mean
    #pragma omp parallel shared(mean, w) private(i, j)
    {
      #pragma omp for
      for (i = 1; i < M - 1; i++)
        for (j = 1; j < N - 1; j++)
          w[i][j] = mean;
    }
  }
```

然后，每个进程计算自己需要计算的范围，这里是按行划分的。注意到，第 0 行和第 M-1 行是不需要计算的，因为它们属于边界节点，温度是固定的。

`my_bound` 函数会根据 `start`、`end`、进程数 `comm_sz` 和进程号 `my_rank` 来计算该进程需要计算的范围。可以处理不能被整除的情况。

```
/* Each process compute my_first_m and my_last_m */
int my_first_m, my_last_m, my_m;
my_bound(1, M - 1, comm_sz, my_rank, &my_first_m, &my_last_m, &my_m);
```

```
void my_bound(int start, int end, int comm_sz, int my_rank, int *my_first_m, int *my_last_m, int *my_m)
{
  if (my_rank == comm_sz - 1) // if I'm the last process
  {
    *my_first_m = start + (end - start) / comm_sz * my_rank;
    *my_last_m = end;  // set my_last_m as M-1
  }
  else
  {
    *my_first_m = start + (end - start) / comm_sz * my_rank;
    *my_last_m = *my_first_m + (end - start) / comm_sz;
  }
  *my_m = *my_last_m - *my_first_m;
}
```

接下来，我们先分析 openmp 版本的 heated plate 迭代部分的代码。

```
#pragma omp parallel shared(u, w) private(i, j)
{
  //  Save the old solution in U.
  #pragma omp for
  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      u[i][j] = w[i][j];

  // Determine the new estimate of the solution at the interior points.
  // The new solution W is the average of north, south, east and west neighbors.
  #pragma omp for
  for (i = 1; i < M - 1; i++)
    for (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;
}

// C and C++ cannot compute a maximum as a reduction operation.

// Therefore, we define a private variable MY_DIFF for each thread.
// Once they have all computed their values, we use a CRITICAL section
// to update DIFF.
diff = 0.0;
#pragma omp parallel shared(diff, u, w) private(i, j, my_diff)
{
  my_diff = 0.0;
  #pragma omp for
  for (i = 1; i < M - 1; i++)
    for (j = 1; j < N - 1; j++)
      if (my_diff < fabs(w[i][j] - u[i][j]))
        my_diff = fabs(w[i][j] - u[i][j]);

  #pragma omp critical
  if (diff < my_diff)
    diff = my_diff;
}
```

首先，迭代开始时，用一个 MxN 的二位数组 `u` 暂存 `w`，这一步是非常耗时的，因为每次迭代都要做一次复制，大量的内存存取造成巨大的开销。考虑到要使用 MPI 并行，如果将整个 `w` 广播给所有的进程，每个进程还要将 `w` 复制到 `u`，开销是非常巨大的，所以我们可以针对这一部分进行优化。我的想法是，master 进程保留 `w` 的值，由于是按行分配计算任务的，所以可以将每个进程需要的 `w` 的部分发送给它，而不是广播整个 `w`。

于是，每个进程可以用一个 `w_buf` 来暂存需要的 `w`， `w_buf` 可以定义为 `double w_buf[my_m + 2][N]`，注意行数为 `my_m + 2` 的原因是计算 `my_m` 行还需要边界的上下两行温度。

但是，这么做还不够，由于在每次迭代更新温度时，是按照上次迭代的温度来更新的，所以如果只用 `w_buf` 一个缓冲区进行收发数据的话，还需要一个 `u_buf` 来暂存接收到的 `w_buf`，再根据 `u_buf` 暂存的值将计算结果保存到 `w_buf`，这又需要将 `w_buf` 复制到 `u_buf`，造成巨大开销。

进一步的，每个进程还可以用一个 `my_w` 来保存自己本次迭代计算的结果，于是就可以直接利用 `w_buf` 来计算，将结果保存到 `my_w` 即可，不需要保存旧值的额外操作。

最后，还可以发现，每个进程计算误差 `my_diff` 和 `my_w[i][j]` 可以合并到一起，这样会将两次双重循环的开销减少到一次。

下面是每个进程都会定义的 `my_w` 和 `w_buf`，分别作为发送缓冲区和接收缓冲区。

```
// set my_w as send buffer, the computation results will be saved in my_w
double my_w[my_m][N];
for (int i = 0; i < my_m; i++)
{
  my_w[i][0] = 100;
  my_w[i][N-1] = 100;
}

// set w_buf as recv buffer
double w_buf[my_m + 2][N];
```

然后进入迭代循环，每次计算开始前，master 进程先把各个进程需要的 `w` 发送给它们，slave 进程用 `w_buf` 接收。当然， `master` 也要计算，将第一个部分 copy 到自己的 `w_buf` 中。

```
// Do not send the whole w
if (my_rank == 0)
{
  for (int slave = 1; slave < comm_sz; slave++)
  {
    int slave_first_m, slave_last_m, slave_m;
    my_bound(1, M - 1, comm_sz, slave, &slave_first_m, &slave_last_m, &slave_m);
    MPI_Send(&w[slave_first_m - 1][0], (slave_m + 2) * N, MPI_DOUBLE, slave, 0, MPI_COMM_WORLD);
  }
  memcpy(&w_buf[0][0], &w[my_first_m - 1][0], sizeof(double) * (my_m + 2) * N);
}
else
{
  MPI_Recv(&w_buf[0][0], (my_m + 2) * N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

然后是计算部分，每个进程将计算的结果保存到 `my_w` 中，同时，某个位置计算完成后，还会更新误差 `my_diff`。

```
// Compute my_w for each process and my_diff at the same time
// Note that my_w[i][j] <=> w_buf[i+1][j]
double my_diff = 0.0;
for (int i = 0; i < my_m; i++)
{
  for (int j = 1; j < N - 1; j++)
  {
    my_w[i][j] = (w_buf[i][j] + w_buf[i+2][j] + w_buf[i+1][j-1] + w_buf[i+1][j+1]) / 4.0;
    my_diff = MAX(my_diff, fabs(my_w[i][j] - w_buf[i+1][j]));
  }
}
```

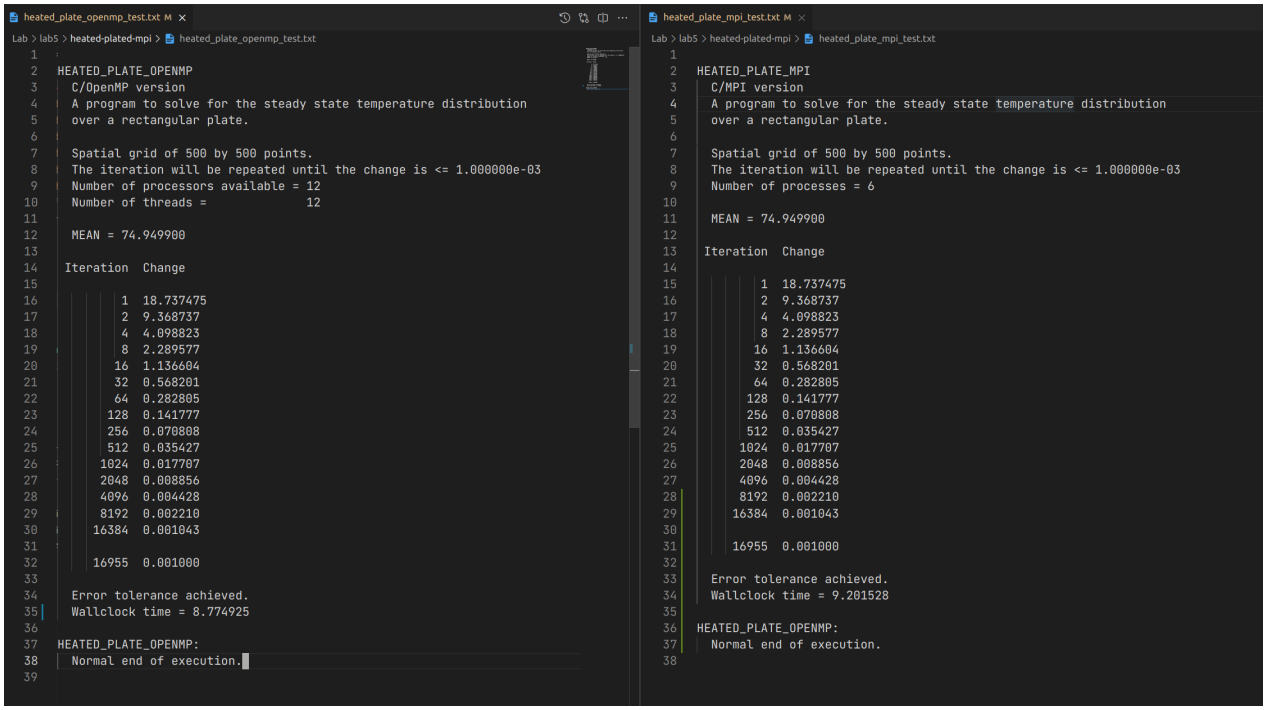
最后是 slave 进程向 master 发送自己的计算结果，这里数组下标位置要细心一点，master 将结果汇总到 `w` 上，这样，下次迭代开始时，又将 `w` 分配给每个进程的 `w_buf`。`w` 更新完成之后，对 `my_diff` 进行 reduce 操作，选取最大的 `my_diff` 到 `diff`。这里使用 `MPI_Allreduce`，因为每个进程都要进行循环条件的判断，误差是否小于阈值。

```
if (my_rank != 0) // if I'm slave, send my_w to master
{
  // printf("I'm process %d, my begin row = %d\n", my_rank, my_first_m);
  MPI_Send(&my_w[0][0], my_m * N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}
else  // if I'm the master, recv my_w from slaves
{
  for (int slave = 1; slave < comm_sz; slave++)
  {
    int slave_first_m, slave_last_m, slave_m;
    my_bound(1, M - 1, comm_sz, slave, &slave_first_m, &slave_last_m, &slave_m);
    MPI_Recv(&w[slave_first_m][0], slave_m * N, MPI_DOUBLE, slave, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
  // master should also copy its own my_w to w
  memcpy(&w[1][0], &my_w[0][0], sizeof(double) * my_m * N);
}

// Reduce the maximum my_diff to diff, every process will get a copy
MPI_Allreduce(&my_diff, &diff, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
```

在 `heated-plate-mpi` 目录下，make 后再执行 `make test`，会依次运行 OpenMP 版本和 MPI 版本的 heated plate。默认开启最大的线程数和进程数，我的 CPU 是 6 核 12 线程的，OpenMP 的线程数为 12，MPI 进程数为 6。

下面是运行的结果。两者迭代的次数和记录的误差完全相同，最后的 Wallclock time 相差只有 0.5 秒，而 OpenMP 的线程数是 MPI 进程数的两倍（可能与 Intel 的超线程技术的实现有关），MPI 版本的性能可以说非常好。



另外，我还将优化前的 MPI 版本的运行结果 backup 了一下，下面是 backup 版本的结果。（优化前指的是，没有用 `w_buf` 作为接收 `w` 必要部分的缓冲区，和没有用 `my_w` 作为计算结果的缓冲区、避免 `w_buf` 的复制，这两部分的优化）

可以看到，没有用缓冲区的墙上时间是 42.5 秒，比我优化后的 9.2 秒慢了接近 5 倍。实际上，我在写代码的时候，先考虑到的是 `w_buf`，再考虑到的是 `my_w`，而前者的优化为程序带来了 2 倍的性能提升，后者为程序带来了 2.5 倍的提升。

由于我发送和接收只用到了 `w`，所以没必要用 MPI pack/unpack 来打包和解包，这对于我优化后的程序并不能带来性能提升，所以没有实现。

heated_plate_mpi_test.text.backup

Lab > lab5 > heated-plated-mpi > heated_plate_mpi_test.text.backup

```
1
2  HEATED_PLATE_MPI
3    C/MPI version
4    A program to solve for the steady state temperature distribution
5    over a rectangular plate.
6
7    Spatial grid of 500 by 500 points.
8    The iteration will be repeated until the change is <= 1.000000e-03
9    Number of processes = 6
10
11   MEAN = 74.949900
12
13   Iteration  Change
14
15           1  18.737475
16           2  9.368737
17           4  4.098823
18           8  2.289577
19          16  1.136604
20          32  0.568201
21          64  0.282805
22         128  0.141777
23         256  0.070808
24         512  0.035427
25        1024  0.017707
26        2048  0.008856
27        4096  0.004428
28        8192  0.002210
29       16384  0.001043
30
31       16955  0.001000
32
33   Error tolerance achieved.
34   Wallclock time = 42.484814
35
36 HEATED_PLATE_OPENMP:
37   Normal end of execution.
38
```

(3) fft_pf 和 fft_openmp 的性能对比

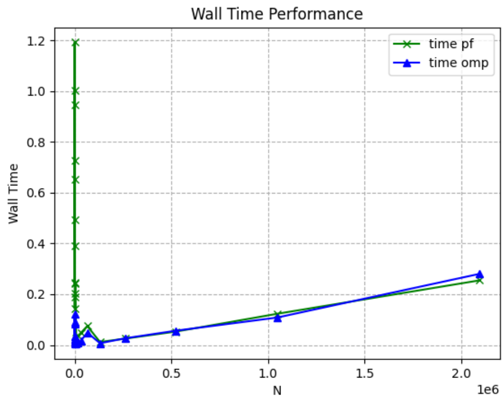
在 `fft` 下执行 `make test`，会在 `asset` 生成误差、运行时间、MFLOPS 的文件。任何并行优化都是在保证程序计算正确的前提下进行的，可以查看生成的文件中的 error 一列，我们的 parallel_for 每次迭代的误差与 openmp 完全相同，说明计算结果没有问题。

下面是可视化后的结果。

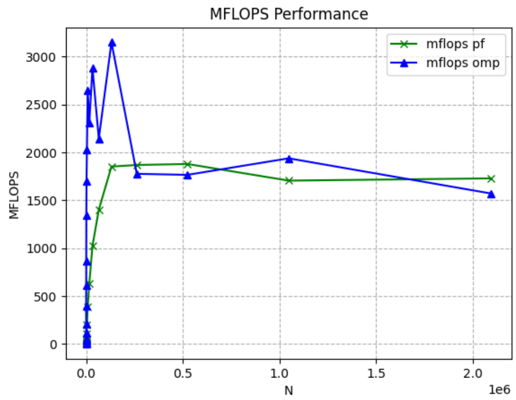
首先是墙上时间，可以看到在 N 较小时（大约是小于 50），我们的 parallel_for 运行时间远大于 OpenMP 的时间，但是在 N > 1000 时，我们的 parallel_for 有着与 OpenMP 几乎相同甚至略优的运行时间。这说明我们的程序在可扩展性上还是很好的，但是在 N 较小时，没有处理好线程数的问题，导致线程开销过大。

然后是 MFLOPS，对于每秒的浮点运算次数，大约在 N < 2000 时，OpenMP 的 MFLOPS 比 parallel_for 要快大约 1.3 ～ 1.5 倍。在 N > 2000 时，parallel_for 和 OpenMP 相差较小，大约相同。

Wall Time



MFLOPS



在 `fft` 下，执行 `make memory-test` ，会使用 valgrind 的 massif 工具对程序进行内存测试。然后执行 `make print` ，会将保存到 `asset` 下的文件打印出来。

parallel_for 的结果：

```
ms_print ./asset/massif.out.pf
--------------------------------------------------------------------------------
Command:            ./bin/fft_pf 12
Massif arguments:   --time-unit=B --stacks=yes --massif-out-file=./asset/massif.out.pf
ms_print arguments: ./asset/massif.out.pf
--------------------------------------------------------------------------------


    MB
205.0^                                                                      :
     |                                                        #:::::@::::::::
     |                                                        #:::::@::::::::
     |                                                        #:::::@::::::::
     |                                                        #:::::@::::::::
     |                                                        #:::::@::::::::@
     |                                              ::::::::  #:::::@::::::::@
     |                                              : ::::::  #:::::@::::::::@
     |                                              : ::::::::#:::::@::::::::@
     |                                          ::: : ::::::: #:::::@::::::::@
     |                                       :: ::: : ::::::: #:::::@::::::::@
     |                                     @@: :::::: ::::::: #:::::@::::::::@
     |                                @@@@@@ : :::::: ::::::: #:::::@::::::::@
     |                             @@@@ @@@@ : :::::: ::::::: #:::::@::::::::@
     |                         @@@@@@@@ @@@@ : :::::: ::::::: #:::::@::::::::@
     |                     @@:@@@@@@@@@ @@@@ : :::::: ::::::: #:::::@::::::::@
     |                 @@@@@ :@@@@@@@@@ @@@@ : :::::: ::::::: #:::::@::::::::@
     |             @@@@@@ @@ :@@@@@@@@@ @@@@ : :::::: ::::::: #:::::@::::::::@
     |         @@@@@ @@@@ @@ :@@@@@@@@@ @@@@ : :::::: ::::::: #:::::@::::::::@
     |     @@@@@ @@@ @@@@ @@ :@@@@@@@@@ @@@@ : :::::: ::::::: #:::::@::::::::@
   0 +----------------------------------------------------------------------->GB
     0                                                                   6.429

Number of snapshots: 64
 Detailed snapshots: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 46 (peak), 53, 63]
```

openmp 的结果：

```
ms_print ./asset/massif.out.omp
--------------------------------------------------------------------------------
Command:            ./bin/fft_openmp
Massif arguments:   --time-unit=B --stacks=yes --massif-out-file=./asset/massif.out.omp
ms_print arguments: ./asset/massif.out.omp
--------------------------------------------------------------------------------


    KB
32.82^ #
     | #
     | #
     | #
     | #
     | #
     | #
     | #
     | #
     | #
     | #
     | #
     | #
     | #:@:::@:::::::::::::::@@:@::::::::::::::::::@::::@:::::@::::::@::::::@:
     | #:@:::@:::: ::: ::::::@ :@:::::: :::::::::::@::::@:::::@::::::@::::::@:
     | #:@:::@:::: ::: ::::::@ :@:::::: :::::::::::@::::@:::::@::::::@::::::@:
     | #:@:::@:::: ::: ::::::@ :@:::::: :::::::::::@::::@:::::@::::::@::::::@:
     | #:@:::@:::: ::: ::::::@ :@:::::: :::::::::::@::::@:::::@::::::@::::::@:
     | #:@:::@:::: ::: ::::::@ :@:::::: :::::::::::@::::@:::::@::::::@::::::@:
     | #:@:::@:::: ::: ::::::@ :@:::::: :::::::::::@::::@:::::@::::::@::::::@:
   0 +----------------------------------------------------------------------->MB
     0                                                                   22.90

Number of snapshots: 86
 Detailed snapshots: [1 (peak), 3, 7, 23, 25, 46, 52, 62, 72, 82]
```

实际在使用 massif 测试的时候，parallel_for 很快就结束了，但 openmp 会一直卡住，我没搞清楚是什么原因，只好 ctrl+c 结束。所以上面 openmp 输出的内存测试的结果应该是有问题的。

(四) 实验结果

实验结果已经在上面展示。

(五) 实验感想

这次实验修复和改进了上次实验的 parallel_for，上次实现的是有 bug 的，不能处理 increment 非 1 的情况，这次的修复经过我多次测试是没有问题的。但是，我的 parallel_for 在任务数较少时不能合理的处理线程派生，性能比 openmp 差很多。

heated plate 的 MPI 版本我自认为优化得比较好，尤其是发送、接收两个缓冲区的设置，以及合理设置发送的数据量、只发生必要的数据，使程序的运行时间比简单直接的版本减少了大约 5 倍。由于程序只需要收发 `w`，所以不需要 MPI pack/unpack，我开始实现的 MPI pack/unpack 版本的性能低于内存访问优化后的 MPI 版本。

总之，我对于这次实验的第二个任务完成得很满意，优化了程序的内存访问，并用 MPI 进行了并行，达到了和 OpenMP 相近的性能。对于 parallel_for 的实现，正确性是得到了保障，但在循环次数较少时，不能合理地设置线程数和任务分配，导致性能较低，这部分需要着重改进。

附录：参考资料

- https://zh.wikipedia.org/zh/%E5%82%85%E9%87%8C%E5%8F%B6%E5%8F%98%E6%8D%A2
- https://zh.wikipedia.org/wiki/%E7%A6%BB%E6%95%A3%E5%82%85%E9%87%8C%E5%8F%B6%E5%8F%98%E6%8D%A2
- https://zh.wikipedia.org/wiki/%E5%BF%AB%E9%80%9F%E5%82%85%E9%87%8C%E5%8F%B6%E5%8F%98%E6%8D%A2、
- https://people.sc.fsu.edu/~jburkardt/c_src/fft_serial/fft_serial.html
- https://people.sc.fsu.edu/~jburkardt/c_src/heated_plate_openmp/heated_plate_openmp.html

- https://stackoverflow.com/questions/6481005/how-to-obtain-the-number-of-cpus-cores-in-linux-from-the-command-line
- https://valgrind.org/docs/manual/ms-manual.html
- https://stackoverflow.com/questions/52063507/how-to-use-valgrinds-massif-out-file-option-correctly