Lab7 - MPI并行应用

实验要求

使用MPI对快速傅里叶变换进行并行化。

问题描述：阅读参考文献中的串行傅里叶变换代码(fft_serial.cpp)，并使用MPI对其进行并行化。

要求：

1. 并行化：使用MPI多进程对fft_serial.cpp进行并行化。为适应MPI的消息传递机制，可能需要对fft_serial代码进行一定调整。
2. 优化：使用MPI_Pack/MPI-Unpack或MPI_Type_create_struct对数据重组后进行消息传递。
3. 分析：

   a) 改变并行规模（进程数）及问题规模（N），分析程序的并行性能；

   b) 通过实验对比，分析数据打包对于并行程序性能的影响；

   c) 使用Valgrind massif工具集采集并分析并行程序的内存消耗。

   注意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。工具参考：Valgrind

实验过程

1. 实现思路

串行傅里叶变换代码中能够并行化的部分是进入函数`cfft2`之后，根据问题规模N的大小调用一次或多次`step`函数，因此实现的思路是，使用多个进程划分`step`函数任务并行执行，进程间通讯则考虑`step`之前需要主进程将计算量分发到子进程，`step`之后主进程接收子进程的计算结果。

关于进程数，里面涉及一个步长单元`mj`，如果一直使用多个进程执行`step`，每个进程划分到的范围可能小于mj，也可能包含多个mj，导致计算结果范围也不在划分范围内，实现复杂度较高，因此根据mj大小，随着mj的增大，会有减少进程量的操作，使每个进程划分范围至少存在一个完整的步长`mj`。

2. 代码实现

`reorderArray`数据重组，在执行傅里叶变换之前，根据FFT过程中数组的元素结合顺序，例如[0, 1, 2, 3, 4, 5, 6, 7]中，第0个元素和第4个元素配对成04，第1个元素与第3个元素配对成13，最终会呈现的顺序其实是[0, 4, 2, 6, 1, 5, 3, 7]，配对如下：

```
0 4 2 6 1 5 3 7
| / | / | / | /
04  26  15  37
 | /    | /
 0426  1537
   |  /
 04261537
```

如此重排能够使数据划分到不同进程时数组元素连续，将原本需要发送两次的数组元素一次发送，降低通讯成本

```
void reorderArray(double* arr, int n) {/*在每次调用cfft2前使用该函数*/
```

```
    double* tmp1 = new double[n * 2];
    double* tmp2 = new double[n * 2];
    for (int i = 0; i < n * 2; i++) {
        tmp2[i] = arr[i];
    }
    for (int mj = 1; mj < n / 2; mj = mj * 2) {
        int index = 0;
        for (int i = 0; i < (n / 2) / mj; i++) {
            for (int j = 0; j < mj; j++) {
                tmp1[index++] = tmp2[(i * mj + j) * 2];
                tmp1[index++] = tmp2[(i * mj + j) * 2 + 1];
            }
            for (int j = 0; j < mj; j++) {
                tmp1[index++] = tmp2[(n / 2 + i * mj + j) * 2];
                tmp1[index++] = tmp2[(n / 2 + i * mj + j) * 2 + 1];
            }
        }
        for (int i = 0; i < n * 2; i++) {
            tmp2[i] = tmp1[i];
        }
    }
    for (int i = 0; i < n * 2; i++) {
        arr[i] = tmp2[i];
    }
    delete[] tmp1;
    delete[] tmp2;
}
```

`step`函数，

```
void step ( int n, int mj, double a[], double b[], double c[],double d[],
            double w[], double sgn,int begin, int end ){
    /*传入begin、end参数表示每个进程的划分范围*/
  double ambr,ambu;
  int j,jw;
  int k;
  double wjw[2];
  int start = 0; // 表示每个子进程中a,b,c,d的索引
  for (j = begin / mj; j < end / mj; j++)
  {
    jw = j * mj;
    wjw[0] = w[jw*2+0]; // cos
    wjw[1] = w[jw*2+1]; // sin
    if ( sgn < 0.0 ) {
      wjw[1] = - wjw[1];
    }
    // a、b来自同一数组的不同位置，c、d来自同一数组的不同位置
    for (k = 0; k < mj; k++)// 在一个mj中连续取a、b值，连续赋c、d值
    {
        c[(start) * 2 + 0] = a[(start) * 2 + 0] + b[(start) * 2 + 0];
        c[(start) * 2 + 1] = a[(start) * 2 + 1] + b[(start) * 2 + 1];

        ambr = a[(start) * 2 + 0] - b[(start) * 2 + 0];
        ambu = a[(start) * 2 + 1] - b[(start) * 2 + 1];
```

```
        d[(start) * 2 + 0] = wjw[0] * ambr - wjw[1] * ambu;
        d[(start) * 2 + 1] = wjw[1] * ambr + wjw[0] * ambu;
        start++;
    }
    // 数组中如此存储 ababab，因此下一个mj范围的索引需要+=mj
    start += mj;
  }
  return;
}
```

`cff2`函数：主进程执行

```
void cfft2(int n, double x[], double y[], double w[], double sgn, int comm_sz)
{/* comm_sz在这表示实际执行任务的进程数 */
    int j, m, mj;
    int tgle;
    int pro_nums = comm_sz;
    int counts = (n / 2) / pro_nums;     // 每个进程划分量
    int begin = 0, end = counts;         // 0号进程执行的起始范围
    m = ( int ) ( log ( ( double ) n ) / log ( 1.99 ) );
    mj = 1;
    tgle = 1;
    for (int i = 1; i < pro_nums; i++) {// 1.分发数据到各个进程
        MPI_Send(&x[i * counts * 2 * 2], counts * 2 * 2, MPI_DOUBLE, i, 1,
MPI_COMM_WORLD);
    }
    step(n, mj, &x[0], &x[mj * 2], &y[0], &y[mj * 2 + 0], w, sgn, begin, end);
    for (int i = 1; i < pro_nums; i++) {// 2.收集各个进程计算结果
        MPI_Recv(&y[i * counts * 2 * 2], counts * 2 * 2, MPI_DOUBLE, i, 1,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    if (n == 2)return;
    for ( j = 0; j < m - 2; j++ )
    {
        mj = mj * 2;
        // 根据mj重新划分,counts翻倍,pro_nums减半
        pro_nums = min((n / 2) / mj, pro_nums);
        counts = (n / 2) / pro_nums;
        end = counts;
        if ( tgle )
        {
            for (int i = 1; i < pro_nums; i++) {// 1.分发数据到各个进程
                MPI_Send(&y[i * counts * 2 * 2], counts * 2 * 2, MPI_DOUBLE, i,
1, MPI_COMM_WORLD);
            }
            step(n, mj, &y[0], &y[mj * 2], &x[0], &x[mj * 2 + 0], w, sgn, begin,
end);
            tgle = 0;
            for (int i = 1; i < pro_nums; i++) {// 2.收集各个进程计算结果
                MPI_Recv(&x[i * counts * 2 * 2], counts * 2 * 2, MPI_DOUBLE, i,
1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
        }
        else
        {
```

```
            for (int i = 1; i < pro_nums; i++) {// 1.分发数据到各个进程
                MPI_Send(&x[i * counts * 2 * 2], counts * 2 * 2, MPI_DOUBLE, i,
1, MPI_COMM_WORLD);
            }
            step(n, mj, &x[0], &x[mj * 2], &y[0], &y[mj * 2 + 0], w, sgn, begin,
end);
            tgle = 1;
            for (int i = 1; i < pro_nums; i ++) {// 2.收集各个进程计算结果
                MPI_Recv(&y[i * counts * 2 * 2], counts * 2 * 2, MPI_DOUBLE, i,
1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
        }
    }
    if ( tgle ){
        ccopy ( n, y, x );
    }
    mj = n / 2;
    end = mj;
    // 只需使用主进程
    step(n, mj, &x[0], &x[n], &y[0], &y[n], w, sgn, begin, end);
    return;
}
```

`cff2_sub`函数：子进程执行

```
void cfft2_sub(int n, double w[], double sgn,int my_rank, int comm_sz)
{   /*  采用与cfft2主函数相似的逻辑构造，使主函数每个send recv都有对应的子进程recv send
        w在进入函数前已经广播到每个进程*/
    int j;
    int m = (int)(log((double)n) / log(1.99));
    int mj = 1;
    int pro_nums = comm_sz;
    int counts = (n / 2) / pro_nums;
    int begin = my_rank * counts;
    int end = begin + counts;
    // 数组x，y随着mj大小变换而变化，每次step需要重新分配空间（或直接分配一个足够大的空间）
    double* x, * y;
    x = new double[2 * 2 * counts];
    y = new double[2 * 2 * counts];
    // 1.接收主进程的计算前数据
    MPI_Recv(&x[0], counts * 2 * 2, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    step(n, mj, &x[0], &x[mj * 2 + 0], &y[0], &y[mj * 2 + 0], w, sgn, begin,
end);
    // 2.发还主进程的计算后数据
    MPI_Ssend(&y[0], counts * 2 * 2, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
    delete[] x;
    delete[] y;
    if (n == 2)return;
    for (j = 0; j < m - 2; j++)
    {
        mj = mj * 2;
        // 根据mj重新划分,counts翻倍,pro_nums减半
        pro_nums = min((n / 2) / mj, pro_nums);
        counts = (n / 2) / pro_nums;
```

```
        begin = my_rank * counts;
        end = begin + counts;
        if (my_rank < pro_nums) {
            x = new double[2 * 2 * counts];
            y = new double[2 * 2 * counts];
            // 1.接收主进程的计算前数据
            MPI_Recv(&x[0], counts * 2 * 2, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            step(n, mj, &x[0 * 2 + 0], &x[mj * 2 + 0], &y[0 * 2 + 0], &y[mj * 2
+ 0], w, sgn, begin, end);
            // 2.发还主进程的计算后数据
            MPI_Ssend(&y[0], counts * 2 * 2, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
            delete[] x;
            delete[] y;
        }
    }
    return;
}
```

3. 优化

实验要求使用数据重组后进行消息传递(MPI_Pack/MPI-Unpack或MPI_Type_create_struct)，但是依照我已经实现的代码（使用`reorderArray`对即将FFT的数据预处理），在每次传播时只需要发送一段连续的数组元素，即只需调用一次`MPI_Send`，在此基础上进行MPI_Pack/MPI-Unpack或MPI_Type_create_struct并没有实际比对意义。

如果使用MPI_Pack，需要对数组x的分段元素进行多次打包，再依次传送，这对原本连续的数组元素打包并无太大作用；如果使用MPI_Type_create_struct，那么只是在一个结构体中存储一个指针，反而在定义存储结构体上有更多开销。二者本质上都是用来传播多个数据项的复杂数据结构的方式，在我已经实现的并行化代码基础上完全没有优化的意义和必要，可将实现的`reorderArray`数据重组函数看作相似的优化处理，在此不做更多处理。

4. 运行结果

编译运行指令：

```
# 串行
g++ fft_serial.cpp -o ser
./ser
# 并行
mpicxx -g -Wall -o par fft_parallel.cpp
mpirun -np 4 ./par
```

在虚拟机上安装Valgrind

```
wget https://sourceware.org/pub/valgrind/valgrind-3.19.0.tar.bz2
tar -jxvf valgrind-3.19.0.tar.bz2

sudo apt-get install automake
sudo apt-get install autoconf

cd valgrind-3.19.0
./autogen.sh
./configure
make -j4
sudo make install
```

使用Valgrind

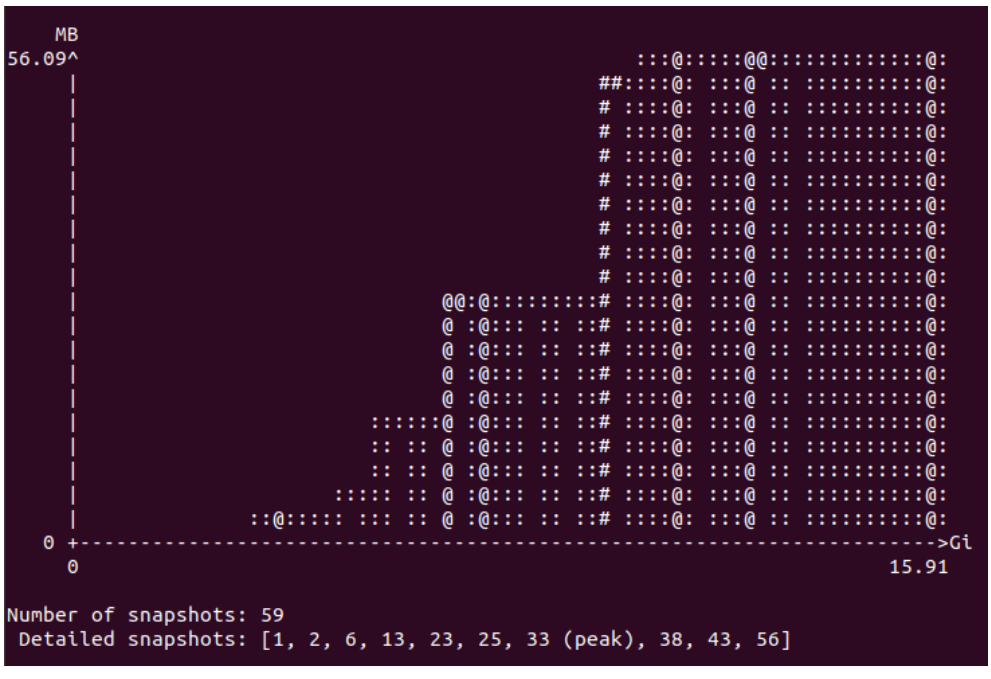
```
# 串行
valgrind --tool=massif --stacks=yes ./ser
# 并行
mpirun -np 4 valgrind --tool=massif --stacks=yes ./par
```

串行版本运行结果

```
 FFT ( FFT ( X(1:N) ) ) == N * X(1:N)

          N      NITS    Error         Time          Time/Call     MFLOPS

          2     10000   7.85908e-17      0.000824      4.12e-08      242.718
          4     10000   1.20984e-16      0.001842      9.21e-08      434.311
          8     10000    6.8208e-17      0.003528     1.764e-07      680.272
         16     10000   1.43867e-16      0.010388     5.194e-07      616.095
         32      1000   1.33121e-16      0.002227    1.1135e-06      718.455
         64      1000   1.77654e-16      0.004591    2.2955e-06      836.419
        128      1000   1.92904e-16       0.01089     5.445e-06      822.773
        256      1000   2.09232e-16      0.024794    1.2397e-05      826.006
        512       100   1.92749e-16      0.005203    2.6015e-05      885.643
       1024       100   2.30861e-16      0.013064     6.532e-05      783.833
       2048       100   2.44762e-16       0.02393    0.00011965      941.412
       4096       100   2.47978e-16      0.057101   0.000285505      860.791
       8192        10   2.57809e-16      0.012726     0.0006363      836.838
      16384        10   2.73399e-16       0.03227     0.0016135      710.803
      32768        10   2.92301e-16      0.053898     0.0026949      911.945
```



并行版本运行结果

此处遇到一些问题，可能是虚拟机配置环境的问题，正常调用检查时未出现错误

```
user@user-virtual-machine:~/lab_7$ mpirun -np 1 valgrind  ./par
==2285== Memcheck, a memory error detector
==2285== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==2285== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==2285== Command: ./par
==2285==
==2285== Warning: ignored attempt to set SIGRT32 handler in sigaction();
==2285==          the SIGRT32 signal is used internally by Valgrind
cr_libinit.c:189 cri_init: sigaction() failed: Invalid argument
==2285==
==2285== Process terminating with default action of signal 6 (SIGABRT)
==2285==    at 0x5A5FE87: raise (raise.c:51)
==2285==    by 0x5A617F0: abort (abort.c:79)
==2285==    by 0x5E1512A: ??? (in /usr/lib/libcr.so.0.5.5)
==2285==    by 0x40108D2: call_init (dl-init.c:72)
==2285==    by 0x40108D2: _dl_init (dl-init.c:119)
==2285==    by 0x40010C9: ??? (in /lib/x86_64-linux-gnu/ld-2.27.so)
==2285==
==2285== HEAP SUMMARY:
==2285==     in use at exit: 0 bytes in 0 blocks
==2285==   total heap usage: 1 allocs, 1 frees, 25 bytes allocated
==2285==
==2285== All heap blocks were freed -- no leaks are possible
==2285==
==2285== For lists of detected and suppressed errors, rerun with: -s
==2285== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
--------------------------------------------------------------------------
mpirun noticed that process rank 0 with PID 0 on node user-virtual-machine exited on signal 6 (Aborted).
--------------------------------------------------------------------------
```

使用mpirun valgrind运行出错：`cri_init: sigaction() failed: Invalid argument`

```
user@user-virtual-machine:~/lab_7$ mpirun -np 2 valgrind --tool=massif --stacks=yes ./par
==2300== Massif, a heap profiler
==2300== Copyright (C) 2003-2017, and GNU GPL'd, by Nicholas Nethercote
==2300== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==2300== Command: ./par
==2300==
==2301== Massif, a heap profiler
==2301== Copyright (C) 2003-2017, and GNU GPL'd, by Nicholas Nethercote
==2301== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==2301== Command: ./par
==2301==
cr_libinit.c:189 cri_init: sigaction() failed: Invalid argument
==2301==
==2301== Process terminating with default action of signal 6 (SIGABRT)
==2301==    at 0x5A55E87: raise (raise.c:51)
==2301==    by 0x5A577F0: abort (abort.c:79)
==2301==    by 0x5E0B12A: ??? (in /usr/lib/libcr.so.0.5.5)
==2301==    by 0x40108D2: call_init (dl-init.c:72)
==2301==    by 0x40108D2: _dl_init (dl-init.c:119)
==2301==    by 0x40010C9: ??? (in /lib/x86_64-linux-gnu/ld-2.27.so)
==2301==
cr_libinit.c:189 cri_init: sigaction() failed: Invalid argument
==2300==
==2300== Process terminating with default action of signal 6 (SIGABRT)
==2300==    at 0x5A55E87: raise (raise.c:51)
==2300==    by 0x5A577F0: abort (abort.c:79)
==2300==    by 0x5E0B12A: ??? (in /usr/lib/libcr.so.0.5.5)
==2300==    by 0x40108D2: call_init (dl-init.c:72)
==2300==    by 0x40108D2: _dl_init (dl-init.c:119)
==2300==    by 0x40010C9: ??? (in /lib/x86_64-linux-gnu/ld-2.27.so)
==2300==
--------------------------------------------------------------------------
mpirun noticed that process rank 0 with PID 0 on node user-virtual-machine exited on signal 6 (Aborted).
--------------------------------------------------------------------------
```

反复搜索各种解决方案尝试修改无果后，反而发现引发另一个错误：正常编译运行该文件会出现每个进程都打印主进程的内容，原本是能正常运行的，出现这个问题不知如何处理，在学校云主机上测试相同文件还是没有问题。

```
user@user-virtual-machine:~/lab_7$ mpirun -np 4 ./par
20 May 2024 10:29:21 PM
           N      NITS    Error         Time          Time/Call     MFLOPS

           2     10000   7.85908e-17      0.006227    3.1135e-07       32.1182
           4     10000   1.20984e-1620 May 2024 10:29:21 PM
           N      NITS    Error         Time          Time/Call     MFLOPS

           2     10000   7.85908e-17      0.003322     1.661e-07       60.2047
           4     10000   1.20984e-1620 May 2024 10:29:21 PM
           N      NITS    Error         Time          Time/Call     MFLOPS

           2     10000   7.85908e-1720 May 2024 10:29:21 PM
           N      NITS    Error         Time          Time/Call     MFLOPS

           2     10000   7.85908e-17      0.003832     1.916e-07       52.1921
           4     10000   1.20984e-16      0.011605    5.8025e-07       68.9358
           8     10000      0.164371      0.005953    2.9765e-07       134.386
           8     10000      0.164371      0.004804     2.402e-07        41.632
           4     10000   1.20984e-16      0.009593    4.7965e-07       83.3941
           8     10000      0.164371      0.012987    6.4935e-07       61.6001
           8     10000      0.164371      0.015833    7.9165e-07       151.582
          16     10000      0.438399      0.023771   1.18855e-06       100.963
          16     10000      0.438399        0.0235     1.175e-06       102.128
          16     10000      0.438399      0.025464    1.2732e-06       94.2507
          16     10000      0.438399      0.084061   4.20305e-06       76.1352
ehpc@61583b2ed2e2:~/data$ mpirun -np 4 ./par
20 May 2024 09:48:04 PM
           N      NITS    Error         Time          Time/Call     MFLOPS

           2     10000   7.85908e-17      0.003452     1.726e-07       57.9374
           4     10000   1.20984e-16      0.043787   2.18935e-06       18.2703
           8     10000      0.164371      0.135014    6.7507e-06       17.7759
          16     10000      0.438399      0.227024   1.13512e-05       28.1909
          32      1000      0.564543      0.036309   1.81545e-05       44.0662
          64      1000      0.501478      0.057593   2.87965e-05       66.6748
         128      1000       0.54418      0.084779   4.23895e-05       105.687
         256      1000       0.55141      0.138094    6.9047e-05       148.305
         512       100      0.542106      0.023433   0.000117165       196.646
        1024       100      0.549289      0.055231   0.000276155       185.403
        2048       100      0.574768      0.138248    0.00069124       162.954
        4096       100      0.571045      0.246726    0.00123363       199.217
        8192        10       0.57436       0.04797    0.0023985       222.005
       16384        10       0.57727      0.122127    0.00610635       187.818
       32768        10      0.573841      0.252135    0.0126067       194.943
20 May 2024 09:48:12 PM
```

结果分析

总结上面存在的未解决问题：

1. 使用Valgrind无法正常运行MPI并行文件
2. 调环境过程导致虚拟机上MPI并行文件输出多余内容
3. 结果正确性未找出代码逻辑问题，目前只有N=2，4时的Error是与并行结果一致的

花费很多时间，但上述问题未能在提交作业前得到解决，只能勉强在云主机上进行性能分析，比较不同进程、不同问题规模的Time/Call，以下数据比较来自云主机上实验，因此可能与上文截图有明显差别。

进程数\规模	8	32	128	512	2048	8192
串行	3.9195e-07	1.9595e-06	9.8345e-06	4.831e-05	0.00023553	0.00114745
1	7.8955e-07	4.178e-06	2.1394e-05	0.000108725	0.00052872	0.00249685
2	3.34825e-06	9.691e-06	3.1146e-05	0.00014819	0.000645515	0.0029311
4	6.3177e-06	1.5575e-05	4.2318e-05	0.00014233	0.000547	0.00247245
8	6.32135e-06	2.6461e-05	6.1288e-05	0.00015594	0.000574755	0.00245895

可以看出并行化后的规模的Time/Call耗时反而比原本串行的长，特别是问题规模较小的时候，并行通信的开销要大于并行带来的计算优势，而在问题规模变大时，如规模=8192一列，同为并行的规模随着进程数变大而耗时变低，才能看出并行化后带来的性能优势。

总结

本次实验的难点主要在于理解FFT串行代码并构思并行化的逻辑并重新熟悉之前使用的MPI，由此也容易自己制造很多bug，在改bug的过程耗费了很多时间，导致后面学习使用Valgrind的时间不足，只能完成串行版本的采集程序运行栈内内存消耗，代码还存在需要改正和简化的地方，但因为时间问题，其他作业deadline堆积，只能在最后两天完成这次实验，只能勉强完成基本要求，提交作业之后还需要着手解决上述问题。