中山大学计算机院本科生实验报告

（2023 学年春季学期）

课程名称：并行程序设计　　**批改人：**

实验	MPI 并行应用	专业（方向）	计算机科学与技术
学号	21307082	姓名	赖耿桂
Email	laigg@mail2.sysu.edu.cn	完成日期	2024.5.12

1. 实验目的（200 字以内）

使用 MPI 对快速傅里叶变换进行并行化，并使用 MPI_Pack/MPI_Unpack 或 MPI_Type_create_struct 对数据重组后进行消息传递来进行优化，比较不同问题规模、不同进程数、有无进行数据重组的性能差异。

2. 实验过程和核心代码（600 字左右，图文并茂）

首先进行并行化，主要对三个最重要的函数进行并行化，分别是 ccopy(数组拷贝)、cfft2(进行傅里叶变换与逆变换)、step(蝶形运算)，此外，还需要对 main 函数进行一定的调整。

在使用 MPI 进行多进程并行时，需要导入头文件 mpi.h 并声明 2 个变量用于记录进程号(myrank)和进程数(process_num)。由于我将 myrank 和 process_num 设置为全局变量，因此在函数中可以直接使用这两个变量。

ccopy 函数

首先，需要根据当前进程的编号，获取当前进程需要拷贝的数组的起始索引和终止索引，然后将该部分的数组 x 的元素拷贝到数组 y 的对应位置上。在拷贝完之后，对于非 0 号进程，使用 MPI_Send 将拷贝好的数组 y 的部分发送给 0 号进程；而 0 号进程则在完成了自己的拷贝任务之后，使用 MPI_Recv 接收来自其他进程的拷贝好的数组 y 的部分。

```
void ccopy(int n, double x[], double y[]) {
    int block = n / process_num;    // 分块大小

    // 对每个进程的数组块进行复制
    for( int i = myrank * block; i < (myrank+1) * block; ++i){
        y[i * 2 + 0] = x[i * 2 + 0];
        y[i * 2 + 1] = x[i * 2 + 1];
    }

    // 接下来进行进程间的通信
    if( myrank ){   // 非0进程，将复制结果发给0号进程的y的对应位置
        MPI_Send(&y[myrank * block * 2], 2 * block, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
    else{   // 0号进程，在对应位置接收来自其他进程发送过来的复制结果
        for(int i = 1; i < process_num; ++i){
            MPI_Recv(&y[i * block * 2], 2 * block, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
    }
    return;
}
```

cfft2 函数

cfft2 函数的代码与串行版本的 cfft2 函数代码基本相同，但是有一点需要特别注意的是，在调用 ccopy 函数或者 step 函数之前，需要使用 MPI_Bcast 广播 0 号进程的数组 x 和 y，这是因为在 MPI 多进程并行编程中，每个进程都有自己独立的内存，不同进程的数组 x 和 y 是不共享的，且在调用 ccopy 函数或者 step 函数后，数组 x 和 y 会被更新，不使用广播来实时更新各个进程的数组 x 和 y 的话会导致计算错误。因此为了保证 FFT 和 IFFT 的正确性，在 cfft2 函数中，每次调用 ccopy 函数或 step 函数之前都要使用 MPII_Bcast 广播 0 号进

程的数组 x 和 y(选择 0 号进程是因为在 main 函数中使用 0 号进程来汇总结果及输出)。

```
1.  void cfft2(int n, double x[], double y[], double w[], double sgn)
2.  {
3.      int j, m, mj, tgle;
4.
5.      m = (int)(log((double)n) / log(1.99));
6.      mj = 1;
7.
8.      tgle = 1;
9.
10.     // 使用MPI_Bcast 将数组x 和y 广播出去
11.     MPI_Bcast(x, 2 * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
12.     MPI_Bcast(y, 2 * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
13.     step(n, mj, &x[0 * 2 + 0], &x[(n / 2) * 2 + 0], &y[0 * 2 + 0], &y[mj * 2 + 0], w, sgn);
14.
15.     if (n == 2){
16.         return;
17.     }
18.
19.     for (j = 0; j < m - 2; j++)
20.     {
21.         mj = mj * 2;
22.         if (tgle){
23.             // 由于前面已经使用step 函数进行了蝶形运算，数组x 和y 是有更新的，为了确保运算的准确性，在每次进行蝶形运算之前都要使用MPI_Bcast 广播更新后的数组x 和y
24.             MPI_Bcast(x, 2 * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
25.             MPI_Bcast(y, 2 * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
26.             step(n, mj, &y[0 * 2 + 0], &y[(n / 2) * 2 + 0], &x[0 * 2 + 0], &x[mj * 2 + 0], w, sgn);
27.             tgle = 0;
28.         }
29.         else{
30.             // 由于前面已经使用step 函数进行了蝶形运算，数组x 和y 是有更新的，为了确保运算的准确性，在每次进行蝶形运算之前都要使用MPI_Bcast 广播更新后的数组x 和y
31.             MPI_Bcast(x, 2 * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
32.             MPI_Bcast(y, 2 * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
33.             step(n, mj, &x[0 * 2 + 0], &x[(n / 2) * 2 + 0], &y[0 * 2 + 0], &y[mj * 2 + 0], w, sgn);
34.             tgle = 1;
35.         }
36.     }
```

```
37.     if (tgle){
38.             // 由于前面已经使用step 函数进行了蝶形运算，数组x 和y 是有更新的，为了确保能够正确拷贝数组，因此需要使用MPI_Bcast 广播更新后的数组x 和y
39.             MPI_Bcast(x, 2 * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
40.             MPI_Bcast(y, 2 * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
41.             ccopy(n, y, x);
42.     }
43.
44.     mj = n / 2;
45.     // 由于前面已经使用ccopy 函数进行了数组拷贝，数组x 和y 是有更新的，为了确保运算的准确性，在每次进行蝶形运算之前都要使用MPI_Bcast 广播更新后的数组x 和y
46.     MPI_Bcast(x, 2*n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
47.     MPI_Bcast(y, 2*n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
48.     step(n, mj, &x[0 * 2 + 0], &x[(n / 2) * 2 + 0], &y[0 * 2 + 0], &y[mj * 2 + 0], w, sgn);
49.
50.     return;
51. }
```

step 函数

对 step 函数的并行化处理，首先要判断问题规模是否达到可以并行的条件(代码中使用 lj/process_num 来判断)，如果问题规模太小，不仅仅会导致并行后效率下降，更有可能导致程序出现错误。因此分为以下两种情况：

- 不适合并行

沿用原来的代码，串行进行蝶形运算。

- 可以并行

对 lj 进行划分，确定每个进程号需要处理的部分，然后根据蝶形运算的原理进行计算(除了外层循环的起点和终止条件不同之外，其他的与串行版本一致)。在每个进程完成计算后，非 0 号进程要将本进程计算好的结果(存储在数组 c 和 d 中)发送给 0 号进程(使用 MPI_Send)，对于 0 号进程则需要接收来自其他进程的计算结果(使用 MPI_Recv)，此外，需要使用 MPI_Barrier 进行阻塞，保证运算结果的完整传输。

```
1.  void step(int n, int mj, double a[], double b[], double c[], double d[], double w[], double sgn){
2.      double ambr, ambu;
3.      int j, ja, jb, jc, jd, jw, k, lj, mj2;
4.      double wjw[2];
5.
6.      mj2 = 2 * mj;
7.      lj = n / mj2;
8.      int block = lj / process_num;   // 分块大小
9.
10.     if( block == 0 ){   // lj < process_num，此时问题规模小于进程数，因为block是整型变量，因此截掉小数点后的部分后值为0
11.         // 此时按照串行的蝶形运算处理
12.         for (j = 0; j < lj; j++){
13.             jw = j * mj;
14.             ja = jw;
15.             jb = ja;
16.             jc = j * mj2;
17.             jd = jc;
18.
19.             wjw[0] = w[jw * 2 + 0];
20.             wjw[1] = w[jw * 2 + 1];
21.
22.             if (sgn < 0.0)
23.             {
24.                 wjw[1] = -wjw[1];
25.             }
26.
27.             for (k = 0; k < mj; k++)
28.             {
29.                 c[(jc + k) * 2 + 0] = a[(ja + k) * 2 + 0] + b[(jb + k) * 2 + 0]; // a和b的实部和
30.                 c[(jc + k) * 2 + 1] = a[(ja + k) * 2 + 1] + b[(jb + k) * 2 + 1]; // a和b的虚部和
31.
32.                 ambr = a[(ja + k) * 2 + 0] - b[(jb + k) * 2 + 0]; // a和b的实部差
33.                 ambu = a[(ja + k) * 2 + 1] - b[(jb + k) * 2 + 1]; // a和b的虚部差
34.
35.                 d[(jd + k) * 2 + 0] = wjw[0] * ambr - wjw[1] * ambu; // 求结果的实部
36.                 d[(jd + k) * 2 + 1] = wjw[1] * ambr + wjw[0] * ambu; // 求结果的虚部
37.             }
38.         }
39.     }
40.     else{
```

```
        // 分块进行蝶形运算
        for(j = myrank * block; j < ( myrank + 1 ) * block; ++j){
            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]; // a 和 b 的实部和
                c[(jc + k) * 2 + 1] = a[(ja + k) * 2 + 1] + b[(jb + k) * 2 + 1]; // a 和 b 的虚部和

                ambr = a[(ja + k) * 2 + 0] - b[(jb + k) * 2 + 0]; // a 和 b 的实部差
                ambu = a[(ja + k) * 2 + 1] - b[(jb + k) * 2 + 1]; // a 和 b 的虚部差

                d[(jd + k) * 2 + 0] = wjw[0] * ambr - wjw[1] * ambu; // 求结果的实部
                d[(jd + k) * 2 + 1] = wjw[1] * ambr + wjw[0] * ambu; // 求结果的虚部
            }

            // 接下来进行进程间的通信
            if( myrank ){   // 非 0 进程将运算结果发送给 0 号进程
                MPI_Send(&c[jc * 2], 2 * mj, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);     // 发送数组 c
                MPI_Send(&d[jd * 2], 2 * mj, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);     // 发送数组 d
            }
            else{   // 0 号进程接收来自其他进程的运算结果
                for(int i = 1; i < process_num; ++i){
                    MPI_Recv(&c[( (i * block) + j ) * mj2 * 2], 2 * mj, MPI_DOUBLE, i, 1, MPI_COMM_
WORLD, MPI_STATUS_IGNORE);
                    MPI_Recv(&d[( (i * block) + j ) * mj2 * 2], 2 * mj, MPI_DOUBLE, i, 2, MPI_COMM_
WORLD, MPI_STATUS_IGNORE);
                }
            }
```

```
            MPI_Barrier(MPI_COMM_WORLD);    // 阻塞，保证其他进程都把运算结果都发送到0 号进程
        }
    }
    return;
}
```

main 函数

对于 main 函数的处理，主要是初始化 MPI 和最后结束 MPI、确认在 0 号进程初始化数组 x、z 以及在 0 号进程计算误差和计时。

```
int main(){
    double ctime, ctime1, ctime2, error, flops, fnm1, mflops, sgn, z0, z1;
    int first, i, icase, it, ln2, n, nits = 10000;
    static double seed;
    double *w, *x, *y, *z;

    // 初始化 MPI
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &process_num);
    if( myrank ==0 ){
        timestamp();
        cout << "\n"
            << "FFT_SERIAL\n"
            << "  C++ version\n"
            << "\n"
            << "  Demonstrate an implementation of the Fast Fourier Transform\n"
            << "  of a complex data vector.\n";
        cout << "\n";
        cout << "  Accuracy check:\n";
        cout << "\n";
        cout << "    FFT ( FFT ( X(1:N) ) ) == N * X(1:N)\n"
            << "\n"
            << "             N      NITS    Error         Time          Time/Call     MFLOPS\n"
            << "\n";
    }

    seed = 331.0;
```

```
    n = 1;

    for (ln2 = 1; ln2 <= 20; ln2++)
    {
        n = 2 * n;

        w = new double[n];
        x = new double[2 * n];
        y = new double[2 * n];
        z = new double[2 * n];

        first = 1;

        for (icase = 0; icase < 2; icase++)
        {
            if( myrank == 0 ){      // 在 0 号进程初始化数组 x 和 z
                if (first){
                    for (i = 0; i < 2 * n; i = i + 2){
                        z0 = ggl(&seed);
                        z1 = ggl(&seed);
                        x[i] = z0;
                        z[i] = z0;
                        x[i + 1] = z1;
                        z[i + 1] = z1;
                    }
                }
                else{
                    for (i = 0; i < 2 * n; i = i + 2){
                        z0 = 0.0;
                        z1 = 0.0;
                        x[i] = z0;
                        z[i] = z0;
                        x[i + 1] = z1;
                        z[i + 1] = z1;
                    }
                }
            }
            cffti(n, w);
            MPI_Barrier(MPI_COMM_WORLD);    // 阻塞，等到所有进程都计算出所需的三角函数值
            if (first)
            {
```

```
                sgn = +1.0;
                cfft2(n, x, y, w, sgn); // 先进行DFT
                MPI_Barrier(MPI_COMM_WORLD);    // 阻塞，保证所有进程都完成计算
                sgn = -1.0;
                cfft2(n, y, x, w, sgn); // 再进行IDFT
                MPI_Barrier(MPI_COMM_WORLD);    // 阻塞，保证所有进程都完成计算
                if( myrank == 0){       // 因为最后的计算结果都会汇总到0号进程，因此在0号进程计算总误差即可
                    fnm1 = 1.0 / (double)n;
                    error = 0.0;
                    for (i = 0; i < 2 * n; i = i + 2){
                        error = error + pow(z[i] - fnm1 * x[i], 2) + pow(z[i + 1] - fnm1 * x[i + 1], 2);
                    }
                    error = sqrt(fnm1 * error);
                    cout << "  " << setw(12) << n << "  " << setw(8) << nits << "  " << setw(12) << error;
                }
                first = 0;
            }
            else
            {
                if( myrank == 0 ){  // 前面的误差汇总到0号进程，要与前面输出的信息相对应，因此在0号进程进行计时
                    ctime1 = cpu_time();
                }
                for (it = 0; it < nits; it++)
                {
                    sgn = +1.0;
                    cfft2(n, x, y, w, sgn);
                    sgn = -1.0;
                    cfft2(n, y, x, w, sgn);
                }
                if( myrank == 0 ){      // 仅在0号进程计时
                    ctime2 = cpu_time();
                    ctime = ctime2 - ctime1;

                    flops = 2.0 * (double)nits * (5.0 * (double)n * (double)ln2);

                    mflops = flops / 1.0E+06 / ctime;

                    cout << "  " << setw(12) << ctime << "  " << setw(12) << ctime / (double)(2 * nits) << "  " << setw(12) << mflops << "\n";
```

```
                }
            }
        }
        if ((ln2 % 4) == 0){
            nits = nits / 10;
        }
        if (nits < 1){
            nits = 1;
        }
        delete[] w;
        delete[] x;
        delete[] y;
        delete[] z;
    }
    if( myrank == 0 ){
        cout << "\n";
        cout << "FFT MPI PARALLEL(UNPACKED):\n";
        cout << "  Normal end of execution.\n";
        cout << "\n";
        timestamp();
    }
    MPI_Finalize();
    return 0;
}
```

使用 MPI_Pack/MPI_Unpack 或 MPI_Type_create_struct 对数据重组

在我的程序中，主要是对 step 函数中的数组 c 和 d 的数据进行重组，我使用一个结构体 STEP_BLOCK 来存储要传输的数组 c 和 d 的数据(指针)以及传输的数组的长度：

```
typedef struct {      // 在step函数中需要传数组c和d，因此考虑把它们两个打包在一起后进行通信
    int length;     // 传输的数组的长度
    double* ArrayC_part;    // 要传输的数组c
    double* ArrayD_part;    // 要传输的数组d
}STEP_BLOCK;
```

然后，参照以前做过的实验，声明并定义函数 Build_MPI_Type 来创建自定义的 MPI 数据类型：

```
1.  void Build_MPI_Type(STEP_BLOCK* Args, MPI_Datatype *MYTYPE, int arrayLength);        // 建立自定义MPI 数据类型
2.
3.  void Build_MPI_Type(STEP_BLOCK* Args, MPI_Datatype *MYTYPE, int arrayLength){
4.      int block_lengths[3] = {1, Args->length, Args->length};  // 每个块的长度
5.      MPI_Datatype types[3] = {MPI_INT, MPI_DOUBLE, MPI_DOUBLE};   // 每个块的数据类型
6.      MPI_Aint start_addr, l_addr, c_addr, d_addr;    // 每个块的地址，用于计算偏移量（要使用结构体的基地址）
7.      MPI_Aint displacements[3];      // 每个块的偏移量
8.      MPI_Get_address(Args, &start_addr);      // 获取基地址
9.
10.     MPI_Get_address(&Args->length, &l_addr);
11.     displacements[0] = l_addr - start_addr;   // 计算 length 的偏移量
12.
13.     MPI_Get_address(Args->ArrayC_part, &c_addr);      // 获取 ArrayC_part 的地址
14.     displacements[1] = c_addr - start_addr;   // 计算 ArrayC_part 的偏移量
15.
16.     MPI_Get_address(Args->ArrayD_part, &d_addr);      // 计算 ArrayD_part 的地址
17.     displacements[2] = d_addr - start_addr;   // 计算 ArrayD_part 的偏移量
18.
19.     MPI_Type_create_struct(3, block_lengths, displacements, types, MYTYPE);     // 创建 MPI 数据类型
20.     MPI_Type_commit(MYTYPE);    // 使其生效
21. }
```

在 step 函数中，声明一个 STEP_BLOCK 类型的变量 stepBlock 来记录该进程的计算结果和数组长度，然后调用 Build_MPI_Type 函数创建 MPI 数据类型 Mytype，并使用它来对打包好的数组 c 和 d 的数据进行传输。注意，在 0 号进程接收到结果后还需要将它们取出来，存储到数组 c 和 d 的对应位置上(仅展示并行部分代码)。

```
1.  // 分块进行蝶形运算
2.  for (j = myrank * block; j < (myrank + 1) * block; ++j)
3.  {
4.      jw = j * mj;
5.      ja = jw;
6.      jb = ja;
7.      jc = j * mj2;
8.      jd = jc;
9.
```

```
10.     wjw[0] = w[jw * 2 + 0];
11.     wjw[1] = w[jw * 2 + 1];
12.
13.     if (sgn < 0.0)
14.     {
15.         wjw[1] = -wjw[1];
16.     }
17.     MPI_Datatype Mytype;  // 自定义的数据类型
18.     STEP_BLOCK stepBlock; // 用于打包 c 和 d 的运算结果
19.     stepBlock.length = mj2;
20.     stepBlock.ArrayC_part = (double *)malloc(mj2 * sizeof(double));
21.     stepBlock.ArrayD_part = (double *)malloc(mj2 * sizeof(double));
22.     for (k = 0; k < mj; k++)
23.     {
24.         c[(jc + k) * 2 + 0] = a[(ja + k) * 2 + 0] + b[(jb + k) * 2 + 0]; // a 和 b 的实部和
25.         c[(jc + k) * 2 + 1] = a[(ja + k) * 2 + 1] + b[(jb + k) * 2 + 1]; // a 和 b 的虚部和
26.
27.         ambr = a[(ja + k) * 2 + 0] - b[(jb + k) * 2 + 0]; // a 和 b 的实部差
28.         ambu = a[(ja + k) * 2 + 1] - b[(jb + k) * 2 + 1]; // a 和 b 的虚部差
29.
30.         d[(jd + k) * 2 + 0] = wjw[0] * ambr - wjw[1] * ambu; // 求结果的实部
31.         d[(jd + k) * 2 + 1] = wjw[1] * ambr + wjw[0] * ambu; // 求结果的虚部
32.
33.         // 将运算结果存到 stepBlock 中
34.         stepBlock.ArrayC_part[k * 2 + 0] = c[(jc + k) * 2 + 0];
35.         stepBlock.ArrayC_part[k * 2 + 1] = c[(jc + k) * 2 + 1];
36.         stepBlock.ArrayD_part[k * 2 + 0] = d[(jd + k) * 2 + 0];
37.         stepBlock.ArrayD_part[k * 2 + 1] = d[(jd + k) * 2 + 1];
38.     }
39.     Build_MPI_Type(&stepBlock, &Mytype, 2 * mj);
40.     // 接下来进行进程间的通信
41.     if (myrank)
42.     { // 非 0 进程将运算结果发送给 0 号进程
43.         // MPI_Send(&c[jc * 2], 2 * mj, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);      // 发送数组 c
44.         // MPI_Send(&d[jd * 2], 2 * mj, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);      // 发送数组 d
45.         MPI_Send(&stepBlock, 1, Mytype, 0, 1, MPI_COMM_WORLD); // 非 0 进程将打包好的结果发送给 0 号进程
46.     }
47.     else
48.     { // 0 号进程接收来自其他进程的运算结果
49.         for (int i = 1; i < process_num; ++i)
50.         {
```

```
            // MPI_Recv(&c[( (i * block) + j ) * mj2 * 2], 2 * mj, MPI_DOUBLE, i, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            // MPI_Recv(&d[( (i * block) + j ) * mj2 * 2], 2 * mj, MPI_DOUBLE, i, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            MPI_Recv(&stepBlock, 1, Mytype, i, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // 接收其他进程发送过来的打包结果

            // 将打包结果存到 c 和 d 对应的位置
            int start = ((i * block) + j) * mj2 * 2;
            int end = start + 2 * mj;
            for (int l = start, p = 0; l < end; ++l, ++p)
            {
                c[l] = stepBlock.ArrayC_part[p];
                d[l] = stepBlock.ArrayD_part[p];
            }
        }
    }
    free(stepBlock.ArrayC_part);
    free(stepBlock.ArrayD_part);
    MPI_Barrier(MPI_COMM_WORLD); // 阻塞，保证其他进程都把运算结果都发送到 0 号进程
    MPI_Type_free(&Mytype);
}
```

3. 实验结果（500 字左右，图文并茂）

➢ 编译

```
mpicxx -g -Wall -o filename filename.cpp
```

(由于我有两个文件 fft_MPI.cpp 即并行但无打包版本，和 fft_MPI_Packed.cpp 即并行且对数据并行打包版本，因此 filename=fft_MPI[或 fft_MPI_Packed])

➢ 运行

```
mpiexec -n p ./filename
```

(p 为进程数量)

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

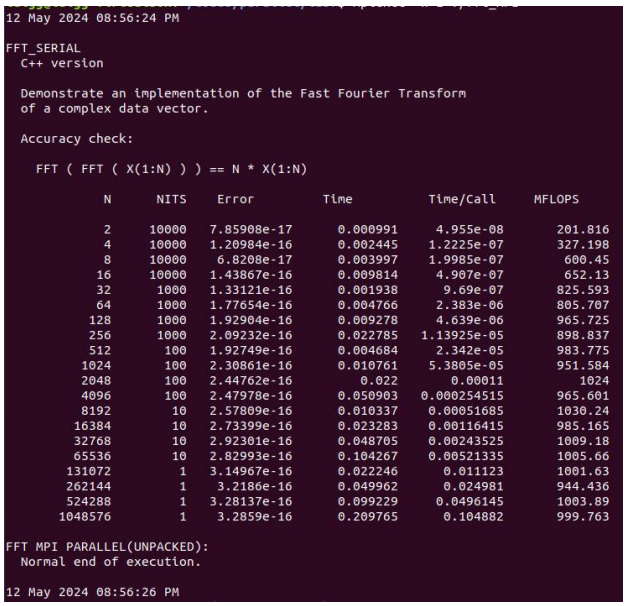
```
valgrind --tool=massif --stacks=yes ./filename
```

执行该命令后会得到一个 massif.out.pid 的输出日志，然后使用以下命令进行打印：

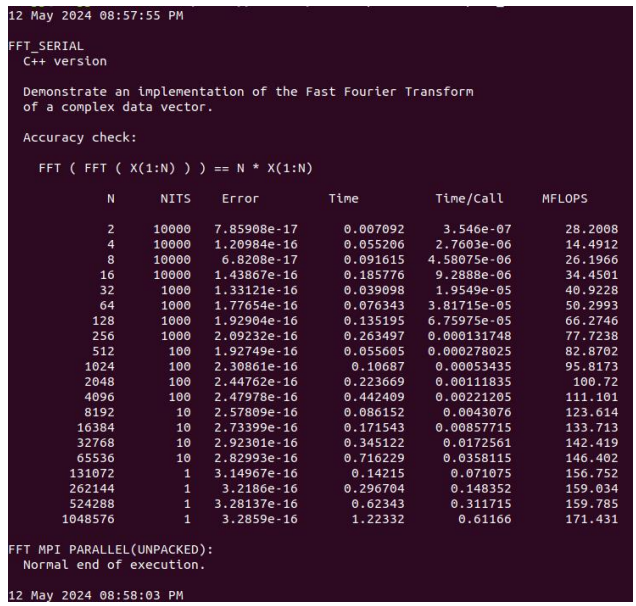
```
ms_print massif.out.pid
```

(pid 为线程号)

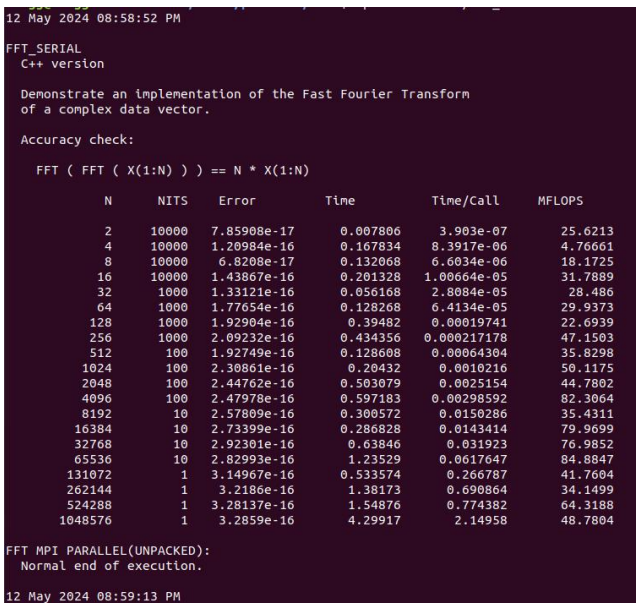
以下为未打包的并行版本在 1、2、4 个进程的输出情况



```
12 May 2024 08:56:24 PM

FFT_SERIAL
  C++ version

  Demonstrate an implementation of the Fast Fourier Transform
  of a complex data vector.

  Accuracy check:

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

           N      NITS    Error         Time          Time/Call     MFLOPS

           2     10000   7.85908e-17      0.000991     4.955e-08       201.816
           4     10000   1.20984e-16      0.002445     1.2225e-07      327.198
           8     10000    6.8208e-17      0.003997     1.9985e-07       600.45
          16     10000   1.43867e-16      0.009814     4.907e-07        652.13
          32      1000   1.33121e-16      0.001938      9.69e-07       825.593
          64      1000   1.77654e-16      0.004766     2.383e-06       805.707
         128      1000   1.92904e-16      0.009278     4.639e-06       965.725
         256      1000   2.09232e-16      0.022785    1.13925e-05      898.837
         512       100   1.92749e-16      0.004684     2.342e-05       983.775
        1024       100   2.30861e-16      0.010761    5.3805e-05       951.584
        2048       100   2.44762e-16         0.022       0.00011          1024
        4096       100   2.47978e-16      0.050903    0.000254515       965.601
        8192        10   2.57809e-16      0.010337     0.00051685      1030.24
       16384        10   2.73399e-16      0.023283     0.00116415      985.165
       32768        10   2.92301e-16      0.048705     0.00243525      1009.18
       65536        10   2.82993e-16      0.104267     0.00521335      1005.66
      131072         1   3.14967e-16      0.022246      0.011123       1001.63
      262144         1    3.2186e-16      0.049962      0.024981       944.436
      524288         1   3.28137e-16      0.099229      0.0496145      1003.89
     1048576         1    3.2859e-16      0.209765      0.104882       999.763

FFT MPI PARALLEL(UNPACKED):
  Normal end of execution.

12 May 2024 08:56:26 PM
```

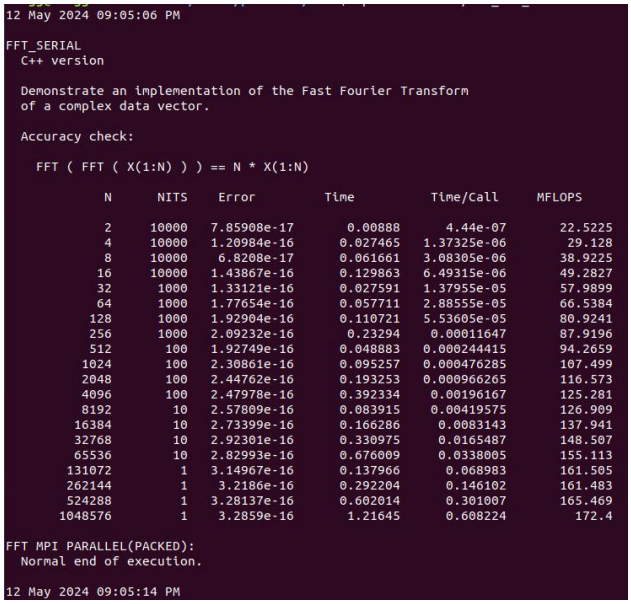


```
12 May 2024 08:57:55 PM

FFT_SERIAL
  C++ version

  Demonstrate an implementation of the Fast Fourier Transform
  of a complex data vector.

  Accuracy check:

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

           N      NITS    Error         Time          Time/Call     MFLOPS

           2     10000   7.85908e-17      0.007092     3.546e-07       28.2008
           4     10000   1.20984e-16      0.055206     2.7603e-06      14.4912
           8     10000    6.8208e-17      0.091615    4.58075e-06      26.1966
          16     10000   1.43867e-16      0.185776     9.2888e-06      34.4501
          32      1000   1.33121e-16      0.039098     1.9549e-05      40.9228
          64      1000   1.77654e-16      0.076343    3.81715e-05      50.2993
         128      1000   1.92904e-16      0.135195    6.75975e-05      66.2746
         256      1000   2.09232e-16      0.263497    0.000131748      77.7238
         512       100   1.92749e-16      0.055605    0.000278025      82.8702
        1024       100   2.30861e-16       0.10687     0.00053435      95.8173
        2048       100   2.44762e-16      0.223669     0.00111835       100.72
        4096       100   2.47978e-16      0.442409     0.00221205      111.101
        8192        10   2.57809e-16      0.086152      0.0043076      123.614
       16384        10   2.73399e-16      0.171543     0.00857715      133.713
       32768        10   2.92301e-16      0.345122      0.0172561      142.419
       65536        10   2.82993e-16      0.716229      0.0358115      146.402
      131072         1   3.14967e-16       0.14215       0.071075      156.752
      262144         1    3.2186e-16      0.296704       0.148352      159.034
      524288         1   3.28137e-16       0.62343       0.311715      159.785
     1048576         1    3.2859e-16       1.22332        0.61166      171.431

FFT MPI PARALLEL(UNPACKED):
  Normal end of execution.

12 May 2024 08:58:03 PM
```

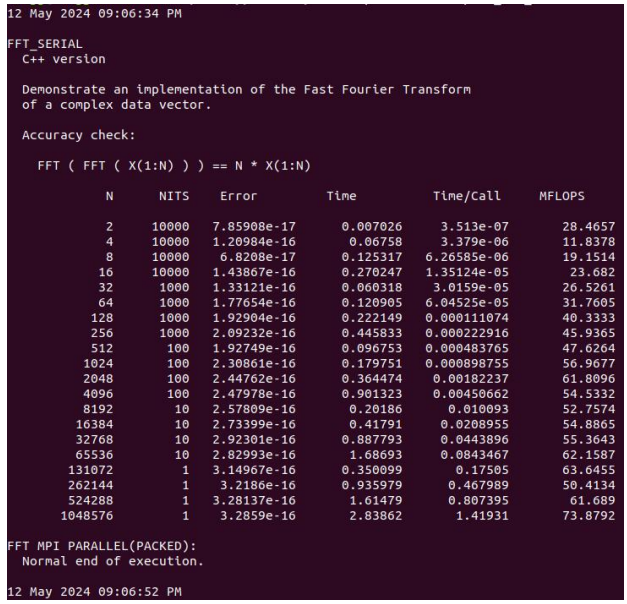


```
12 May 2024 08:58:52 PM

FFT_SERIAL
  C++ version

  Demonstrate an implementation of the Fast Fourier Transform
  of a complex data vector.

  Accuracy check:

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

           N      NITS    Error         Time          Time/Call     MFLOPS

           2     10000   7.85908e-17      0.007806     3.903e-07       25.6213
           4     10000   1.20984e-16      0.167834     8.3917e-06      4.76661
           8     10000    6.8208e-17      0.132068     6.6034e-06      18.1725
          16     10000   1.43867e-16      0.201328    1.00664e-05      31.7889
          32      1000   1.33121e-16      0.056168     2.8084e-05       28.486
          64      1000   1.77654e-16      0.128268     6.4134e-05      29.9373
         128      1000   1.92904e-16       0.39482     0.00019741      22.6939
         256      1000   2.09232e-16      0.434356    0.000217178      47.1503
         512       100   1.92749e-16      0.128608     0.00064304      35.8298
        1024       100   2.30861e-16       0.20432      0.0010216      50.1175
        2048       100   2.44762e-16      0.503079      0.0025154      44.7802
        4096       100   2.47978e-16      0.597183     0.00298592      82.3064
        8192        10   2.57809e-16      0.300572      0.0150286      35.4311
       16384        10   2.73399e-16      0.286828      0.0143414      79.9699
       32768        10   2.92301e-16       0.63846       0.031923      76.9852
       65536        10   2.82993e-16       1.23529      0.0617647      84.8847
      131072         1   3.14967e-16      0.533574       0.266787      41.7604
      262144         1    3.2186e-16       1.38173       0.690864      34.1499
      524288         1   3.28137e-16       1.54876       0.774382      64.3188
     1048576         1    3.2859e-16       4.29917        2.14958      48.7804

FFT MPI PARALLEL(UNPACKED):
  Normal end of execution.

12 May 2024 08:59:13 PM
```

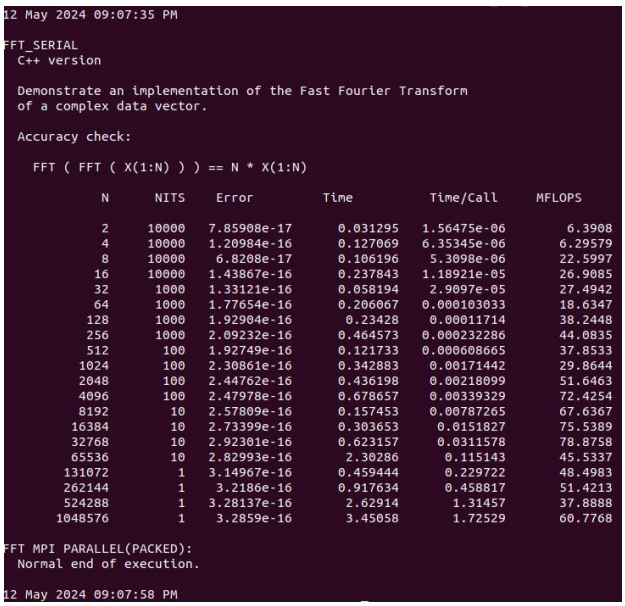
以下为打包的并行版本在 1、2、4 个进程的输出情况



```
12 May 2024 09:05:06 PM

FFT_SERIAL
  C++ version

  Demonstrate an implementation of the Fast Fourier Transform
  of a complex data vector.

  Accuracy check:

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

           N      NITS    Error         Time          Time/Call     MFLOPS

           2     10000   7.85908e-17      0.00888      4.44e-07      22.5225
           4     10000   1.20984e-16     0.027465   1.37325e-06       29.128
           8     10000    6.8208e-17     0.061661   3.08305e-06      38.9225
          16     10000   1.43867e-16     0.129863   6.49315e-06      49.2827
          32      1000   1.33121e-16     0.027591   1.37955e-05      57.9899
          64      1000   1.77654e-16     0.057711   2.88555e-05      66.5384
         128      1000   1.92904e-16     0.110721   5.53605e-05      80.9241
         256      1000   2.09232e-16      0.23294    0.00011647      87.9196
         512       100   1.92749e-16     0.048883   0.000244415      94.2659
        1024       100   2.30861e-16     0.095257   0.000476285      107.499
        2048       100   2.44762e-16     0.193253   0.000966265      116.573
        4096       100   2.47978e-16     0.392334    0.00196167      125.281
        8192        10   2.57809e-16     0.083915    0.00419575      126.909
       16384        10   2.73399e-16     0.166286     0.0083143      137.941
       32768        10   2.92301e-16     0.330975     0.0165487      148.507
       65536        10   2.82993e-16     0.676009     0.0338005      155.113
      131072         1   3.14967e-16     0.137966      0.068983      161.505
      262144         1    3.2186e-16     0.292204      0.146102      161.483
      524288         1   3.28137e-16     0.602014      0.301007      165.469
     1048576         1    3.2859e-16      1.21645      0.608224        172.4

FFT MPI PARALLEL(PACKED):
  Normal end of execution.

12 May 2024 09:05:14 PM
```



```
12 May 2024 09:06:34 PM

FFT_SERIAL
  C++ version

  Demonstrate an implementation of the Fast Fourier Transform
  of a complex data vector.

  Accuracy check:

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

           N      NITS    Error         Time          Time/Call     MFLOPS

           2     10000   7.85908e-17     0.007026     3.513e-07      28.4657
           4     10000   1.20984e-16      0.06758      3.379e-06      11.8378
           8     10000    6.8208e-17     0.125317   6.26585e-06      19.1514
          16     10000   1.43867e-16     0.270247   1.35124e-05       23.682
          32      1000   1.33121e-16     0.060318    3.0159e-05      26.5261
          64      1000   1.77654e-16     0.120905   6.04525e-05      31.7605
         128      1000   1.92904e-16     0.222149   0.000111074      40.3333
         256      1000   2.09232e-16     0.445833   0.000222916      45.9365
         512       100   1.92749e-16     0.096753   0.000483765      47.6264
        1024       100   2.30861e-16     0.179751   0.000898755      56.9677
        2048       100   2.44762e-16     0.364474    0.00182237      61.8096
        4096       100   2.47978e-16     0.901323    0.00450662      54.5332
        8192        10   2.57809e-16      0.20186      0.010093      52.7574
       16384        10   2.73399e-16      0.41791     0.0208955      54.8865
       32768        10   2.92301e-16     0.887793     0.0443896      55.3643
       65536        10   2.82993e-16      1.68693     0.0843467      62.1587
      131072         1   3.14967e-16     0.350099       0.17505      63.6455
      262144         1    3.2186e-16     0.935979      0.467989      50.4134
      524288         1   3.28137e-16      1.61479      0.807395       61.689
     1048576         1    3.2859e-16      2.83862       1.41931      73.8792

FFT MPI PARALLEL(PACKED):
  Normal end of execution.

12 May 2024 09:06:52 PM
```



```
12 May 2024 09:07:35 PM

FFT_SERIAL
  C++ version

  Demonstrate an implementation of the Fast Fourier Transform
  of a complex data vector.

  Accuracy check:

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

           N      NITS    Error         Time          Time/Call     MFLOPS

           2     10000   7.85908e-17     0.031295   1.56475e-06       6.3908
           4     10000   1.20984e-16     0.127069   6.35345e-06      6.29579
           8     10000    6.8208e-17     0.106196    5.3098e-06      22.5997
          16     10000   1.43867e-16     0.237843   1.18921e-05      26.9085
          32      1000   1.33121e-16     0.058194    2.9097e-05      27.4942
          64      1000   1.77654e-16     0.206067   0.000103033      18.6347
         128      1000   1.92904e-16      0.23428    0.00011714      38.2448
         256      1000   2.09232e-16     0.464573   0.000232286      44.0835
         512       100   1.92749e-16     0.121733   0.000608665      37.8533
        1024       100   2.30861e-16     0.342883    0.00171442      29.8644
        2048       100   2.44762e-16     0.436198    0.00218099      51.6463
        4096       100   2.47978e-16     0.678657    0.00339329      72.4254
        8192        10   2.57809e-16     0.157453    0.00787265      67.6367
       16384        10   2.73399e-16     0.303653     0.0151827      75.5389
       32768        10   2.92301e-16     0.623157     0.0311578      78.8758
       65536        10   2.82993e-16      2.30286      0.115143      45.5337
      131072         1   3.14967e-16     0.459444      0.229722      48.4983
      262144         1    3.2186e-16     0.917634      0.458817      51.4213
      524288         1   3.28137e-16      2.62914       1.31457      37.8888
     1048576         1    3.2859e-16      3.45058       1.72529      60.7768

FFT MPI PARALLEL(PACKED):
  Normal end of execution.

12 May 2024 09:07:58 PM
```

不同问题规模的性能分析：

可以看到在随着 N 的增大，总体上每 call 所用的时间是增加的，一方面，问题规模增大意味着要处理的数据会增加(且是呈指数级增长)，另一方面，设备的硬件条件的限制(仅 4 个处理器)也是随着问题规模 N 的增大，每 call 所用时间呈指数级增加的原因。

不同进程数的性能分析：

可以看到，不管是未打包版本还是打包版本，进程数增加后效率总体上不增反降，这并不符合常理，但是通过分析我的代码，可能是以下原因造成的：

①在我的代码中基本上是使用 MPI 点对点通信，其相较于 MPI 集合通信效率更低，且我的通信多在循环内部，随着进程数的增加和问题规模的增大，通信次数增加，通信开销也进一步增大。

②在我的代码中多次使用了 MPI_Barrier 进行阻塞，这就导致在进程数增大时，为了让多个进程同步阻塞的情况更多，这也是导致进程数越多时效率下降的原因之一。

数据打包的影响：

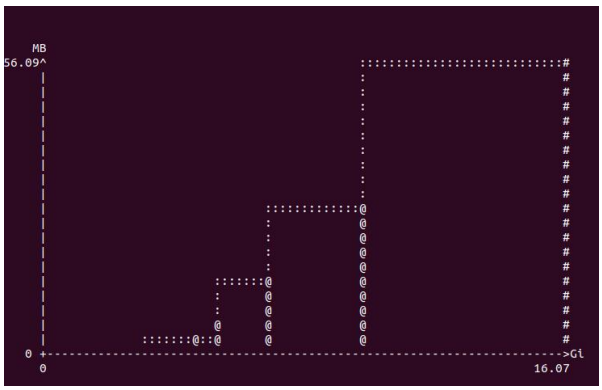
正常来说，数据打包可以提高并行性能，但是我的实验结果展示出来的却是数据打包的并行性能还不如不进行数据打包的并行性能。对于这一反常现象，我认为可能是以下原因造成的：

①0 号进程在接收到其他进程的打包发送过来的结果后，还需要再经过一个循环来讲打包的结果放到数组 c 和 d 中，这带来了额外的运算量。

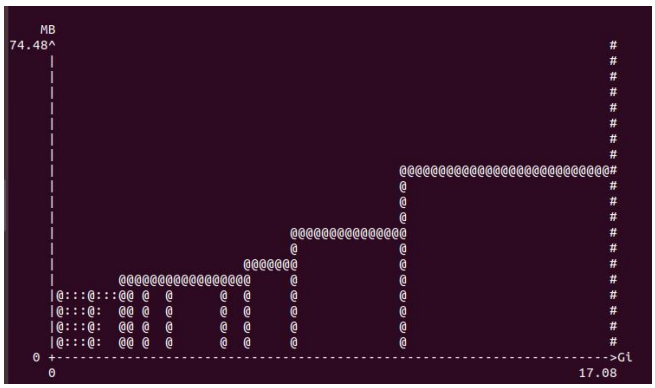
②在传输打包好的数据时依然使用 MPI 点对点通信，相比于使用 MPI 集合通信，这是效率低下的原因之一。

使用 Valgrind massif 工具集采集并分析并行程序的内存消耗：

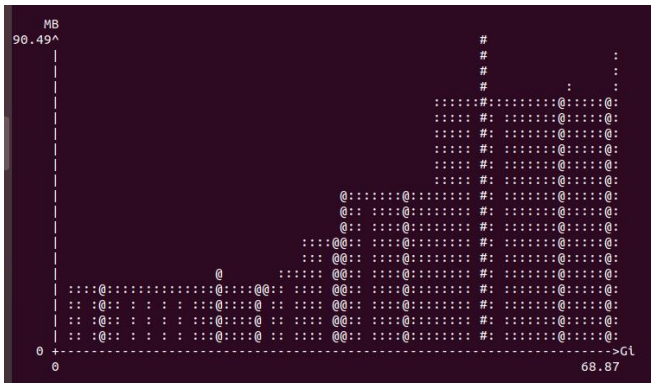
- 串行 fft 的内存消耗：



- 并行数据未打包 fft 的内存消耗：



➢ 并行数据打包 fft 的内存消耗：



将串行 fft 和并行 fft 的数据打包与未打包版本在串行 fft 的时间范围之内(0-16.7Gi)相比，可以看到，使用并行多进程 fft 的内存消耗总体上来说是要小一些的，这可能是因为：

①并行程序将任务分割成多个子任务，这些子任务可以在不同的处理器或核心上同时执行。这意味着每个子任务可能只需要分配较小的内存空间(起码比串行的小)，而不是整个任务所需的全部内存。

②在串行程序中，所有任务都是顺序执行的，因此可能需要在内存中保留更多的状态信息。而在并行程序中，由于任务是同时执行的，因此可以减少这种状态信息的保留。

此外我还注意到使用数据打包的并行版本越往后内存消耗越大，这可能是因为：我使用的结构体需要维护两个指针，且随着时间推移，问题规模 N 增大，两个指针指向的数组也越大，消耗的内存也越大。

4. 实验感想（300 字左右）

在此次实验中，我遇到的比较大的问题是不同进程之间的通信以及一些数据的同步的问题(导致的后果：数据出错、段错误等)，这困扰了我很久，在与其他同学讨论了许久之后问题才得到解决。这反映出我对 MPI 多进程并行编程的理解还不够透彻，这也导致此次实验的效果很差，因此需要对理论知识进行一定的回顾并增加一定的编程训练。

此外，通过此次实验我初步学习了解了如何使用 Valgrind massif 工具集采集并分析并行程序的内存消耗，这对我后续的学习和实践将有很大的帮助。