# Parallel-Programming Task7

刘森元, 21307289

中山大学计算机学院

Codes on https://github.com/Myocardial-infarction-Jerry/Parallel-Programming/tree/main/Task7.

Project built by CMake.

```
> cd Task7
> cmake . && make
> mpirun -np 4 bin/fft_mpi
```

## 1 Environment

Apple M1 Pro

macOS Sonoma 14.4.1

## 2 Task：MPI 并行应用

> **Note**
>
> 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、fft_openmp.c)，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。

## 3 Theory

### 3.1 并行化概念

1. **数据分解**:

- 在FFT中，数据可以被分解为多个独立的片段，这些片段可以并行处理。在此代码中，原始数据向量被分为由多个进程处理的部分。

2. **任务分解**：
   - FFT算法的每一个步骤或迭代可以被视为一个独立的任务，这些任务可以被分配到不同的进程上并行执行。
3. **使用MPI进行通信**：
   - 使用`MPI_Comm_rank`和`MPI_Comm_size`来确定进程的数量和每个进程的ID。
   - 使用`MPI_Bcast`在所有进程中广播数据，确保每个进程都有最新的数据视图。
   - 使用`MPI_Pack`和`MPI_Unpack`以及`MPI_Send`和`MPI_Recv`在进程之间发送和接收数据。

### 3.2 并行FFT的关键步骤

1. **位反转置换**：
   - 数据的初始重排是根据位反转（bit-reversal）的顺序来进行的，这是FFT算法中的典型步骤，以确保输入数据的正确顺序。
2. **蝶形计算**：
   - FFT的核心是蝶形计算，涉及复数的加减和旋转（通过乘以复数旋转因子）。这部分是在所有进程中广播和本地执行后，结果需要被集中处理。
3. **同步与错误检测**：
   - 在FFT的每一个阶段后进行同步，确保所有进程的计算在进入下一阶段前都已完成。最后计算误差，验证FFT的正确性。

### 3.3 性能与优化

- **并行效率**：在理想情况下，增加进程的数量应该减少程序的总运行时间，但由于通信和数据同步的开销，实际的加速比可能会受到影响。
- **负载均衡**：合理分配每个进程的工作量是提高并行程序效率的关键。这个示例中尝试通过在不同的迭代步骤中动态调整任务量（通过调整`nits`）来管理负载。

## 4 Code

> **Caution**
>
> 源代码详见 parallel_for.cpp

```cpp
void FFT(std::vector<std::complex<double>> &a, int sign = 1) {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int n = a.size();
    int ln2 = std::log2(n);

```

```
    if (rank == 0) {
        for (int i = 0; i < n; ++i) {
            int rev = reverse(i, ln2);
            if (i < rev)
                std::swap(a[i], a[rev]);
        }
    }

    for (int s = 1; s <= ln2; ++s) {
        int m = 1 << s;
        int m2 = m >> 1;
        std::complex<double> wm(cos(-2 * PI * sign / m), sin(-2 * PI * sign / m));

        MPI_Bcast(a.data(), n, MPI_DOUBLE_COMPLEX, 0, MPI_COMM_WORLD);
        int bufferSize = 0, position = 0;
        MPI_Pack_size(n, MPI_CXX_DOUBLE_COMPLEX, MPI_COMM_WORLD, &bufferSize);
        auto buffer = new char[bufferSize];

        for (int j = rank * m; j < n; j += size * m) {
            std::complex<double> w(1, 0);
            for (int k = 0; k < m2; ++k) {
                std::complex<double> t = w * a[j + k + m2];
                std::complex<double> u = a[j + k];
                a[j + k] = u + t;
                a[j + k + m2] = u - t;
                w *= wm;
            }

            MPI_Pack(a.data() + j, m, MPI_DOUBLE_COMPLEX, buffer, bufferSize, &position,
MPI_COMM_WORLD);
        }

        if (rank)
            MPI_Send(buffer, bufferSize, MPI_PACKED, 0, 0, MPI_COMM_WORLD);

        if (rank == 0) {
            for (int i = 0; i < size; ++i) {
                if (i)
                    MPI_Recv(buffer, bufferSize, MPI_PACKED, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

                position = 0;

                for (int j = i * m; j < n; j += size * m)
                    MPI_Unpack(buffer, bufferSize, &position, a.data() + j, m,
MPI_DOUBLE_COMPLEX, MPI_COMM_WORLD);
            }
        }

        delete[] buffer;
    }

```

```
    if (rank == 0 && sign == -1)
        for (auto &x : a)
            x /= n;
}
```

# 5 Result

以 8 核心默认调度模式下为例

```
20 May 2024 11:13:15 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      0.000426      2.13e-08       469.484
             4     10000   1.20984e-16      0.001262      6.31e-08       633.914
             8     10000    6.8208e-17      0.002646     1.323e-07       907.029
            16     10000   1.43867e-16      0.006829    3.4145e-07        937.18
            32      1000   1.33121e-16      0.001495     7.475e-07       1070.23
            64      1000   1.77654e-16       0.00352      1.76e-06       1090.91
           128      1000   1.92904e-16      0.007249    3.6245e-06       1236.03
           256      1000   2.09232e-16      0.016858     8.429e-06       1214.85
           512       100   1.92749e-16      0.003607    1.8035e-05       1277.52
          1024       100   2.31209e-16      0.008184     4.092e-05       1251.22
          2048       100   2.44501e-16      0.017886     8.943e-05       1259.53
          4096       100   2.47659e-16      0.038805   0.000194025       1266.64
          8192        10   2.57125e-16      0.009428     0.0004714       1129.57
         16384        10    2.7363e-16      0.019859    0.00099295       1155.02
         32768        10   2.92413e-16        0.0393      0.001965       1250.69
         65536        10   2.83355e-16      0.083041    0.00415205       1262.72
        131072         1   3.14231e-16      0.020243     0.0101215       1100.74
        262144         1    3.216e-16       0.037944      0.018972       1243.57
        524288         1   3.28266e-16      0.080407     0.0402035       1238.88
       1048576         1   3.28448e-16      0.175758      0.087879        1193.2

FFT_MPI:
  Normal end of execution.

20 May 2024 11:13:17 PM
```

1. **数值精度和误差**:

   - 在FFT的过程中，由于使用了浮点运算，可能会累积舍入误差。这些误差随着数据大小的增加而累积，但由于FFT算法的数学性质，这些误差保持在一个相对较低的水平（在本例中都是1e-16级别）。

2. **计算时间和数据大小**:

   - 计算时间随数据大小呈对数增长，这符合FFT的计算复杂度O(N log N)。当N值加倍时，计算时间并不是简单地加倍，而是增加了额外的对数因子。这解释了为什么计算时间增长速度快于N的直线增长。

3. **每次调用时间**:

   - 每次调用的时间主要受到数据大小和FFT算法效率的影响。随着N的增加，每次FFT调用处理的数据点更多，因此时间增加。

4. **MFLOPS的变化**:

   - MFLOPS是衡量程序性能的指标，计算为每秒执行的百万次浮点运算。理想情况下，随着处理器优化和缓存效果的提升，MFLOPS应当随着N的增加而增加。然而，在实际中，当N很大时，数据可能无法完全适配CPU缓存，导致数据传输成为瓶颈，MFLOPS可能不会显著增加，甚至有所下降。

5. **硬件和环境因素**:

   - 处理器的性能、内存带宽、系统负载以及编译器优化等因素都可以影响FFT的执行时间和效率。