中山大學
SUN YAT-SEN UNIVERSITY

并行程序设计与算法实验

Lab7-MPI 并行应用

姓 名 马岱

学 号 22336180

学 院 计算机学院

专 业 计算机科学与技术

2025 年 5 月 7 日

1 实验目的

- 验证并行 FFT 的加速效果与可扩展性
- 评估数据打包对通信性能的优化作用
- 分析并行规模对内存消耗的影响

2 实验内容

- **串行 FFT 分析与并行化改造：**
  - 阅读并理解提供的串行傅里叶变换代码 (fft_serial.cpp)。
  - 使用 MPI (Message Passing Interface) 对串行 FFT 代码进行并行化改造，可能需要对原有代码结构进行调整以适应并行计算模型。
- **MPI 数据通信优化：**
  - 研究并应用 MPI 数据打包技术（例如使用 `MPI_Pack`/`MPI_Unpack` 或 `MPI_Type_create_struct`）来对通信数据进行重组，以优化消息传递效率。
- **程序性能与内存分析：**
  - **性能分析：**
    - 分析在不同并行规模（即不同的进程数量）以及不同问题规模（即输入数据 N 的大小）条件下，并行 FFT 程序的性能表现（例如，通过计算加速比和并行效率来衡量）。
    - 分析数据打包技术对于并行程序整体性能的具体影响。
  - **内存消耗分析：**
    - 使用 Valgrind 工具集中的 Massif 工具来采集并分析并行程序在不同配置下的内存消耗情况。
    - 在 Valgrind 命令中增加 `--stacks=yes` 参数以采集程序运行时栈内内存的消耗情况。
    - 利用 `ms_print` 工具将 Massif 输出的日志 (massif.out.pid) 可视化或转换为可读格式，分析内存消耗随程序运行时间的变化，特别是关注峰值内存消耗。

3 实验结果与分析

3.1 并行 FFT 的实现与正确性验证

本并行 FFT 程序基于 MPI 实现。开始时首先调用 `MPI_Init` 初始化 MPI 环境，并通过 `MPI_Comm_rank` 与 `MPI_Comm_size` 获取当前进程的编号及进程总数。主进程（即 `rank = 0`）随后输出程序基本信息与性能测试表头。

```
// Initialize MPI
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

if (rank == 0) {
    timestamp();
    cout << "FFT_SERIAL\n";
    // ... print headers ...
}
```

接着对不同规模的序列 $N = 2^1, 2^2, \ldots, 2^{20}$ 依次进行测试。对于每个 N，首先检查其是否能被进程总数整除，若不能整除则跳过该规模的测试。若条件满足，则根据进程数划分子问题大小 `local_n`，并为全局数组与本地数组动态分配内存空间。

```
if (n % size != 0) { ... continue; }
local_n = n / size;

// 分配全局数组
w = new double[2 * n];
x = new double[2 * n];
y = new double[2 * n];
z = new double[2 * n];

// 分配本地数组
local_x = new double[2 * local_n];
local_y = new double[2 * local_n];
local_z = new double[2 * local_n];
```

所有进程均分配大小为 $2n$ 的全局数组（用于 `MPI_Scatter/Allgather` 缓冲），而每个进程只对等分后的子数组 `local_x`, `local_y`, `local_z` 进行实际计算。双倍长度（`2*n`）是因为使用实部和虚部交错存储。

接下来调用 cffti 函数，对长度为 N 的序列预计算旋转因子（即正弦与余弦表）。随后程序进入两个阶段：精度验证阶段与性能测试阶段。

在精度验证阶段（设 icase = 0），主进程生成一组随机复数序列，利用 MPI_Scatter 将数据分发至各个进程进行正向 FFT，再通过 MPI_Allgather 聚合结果，并在主进程执行一次逆向 FFT。程序最后比较原始数据与逆变换后的数据之间的差异，用以验证数值精度。

```
if (first) {
    // 主进程生成随机序列 z->x
    for (i = 0; i < 2*n; i+=2) {...}
}
// 数据分发
MPI_Scatter(x, 2*local_n, MPI_DOUBLE, local_x, 2*local_n, MPI_DOUBLE, 0,
↪  MPI_COMM_WORLD);
// 广播旋转因子
MPI_Bcast(w, 2*n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

// 正变换
sgn = +1.0;
MPI_Allgather(local_y, 2*local_n, MPI_DOUBLE, y, 2*local_n, MPI_DOUBLE,
↪  MPI_COMM_WORLD);
if (rank==0) cfft2(n, x, y, w, sgn);

// 逆变换
MPI_Scatter(y, ...);
MPI_Allgather(local_x, ...);
if (rank==0) cfft2(n, y, x, w, -sgn);

// 误差计算并打印
if (rank==0) {... compute sqrt error ...}
first = 0;
```

本阶段用于验证 FFT 算法的正确性：主进程生成随机复数序列 X，并执行一次正向 FFT 和一次反向 FFT。理论上应满足 $\mathrm{IFFT}(\mathrm{FFT}(X)) = NX$，误差计算使用二范数度量数值偏离。MPI 通信调用：先用 MPI_Scatter 分发输入，再全局收集后在主进程执行全局 FFT，接着反向通信重组数据并执行反向 FFT。

在性能测试阶段（设 icase = 1），程序重复执行若干次正向与逆向 FFT，通过 MPI_Wtime 测量总运行时间，进而计算浮点操作性能指标 MFLOPS。

```
MPI_Barrier(MPI_COMM_WORLD);
start_time = MPI_Wtime();
for (it = 0; it < nits; it++) {
    // 正向 FFT 通信 + 计算
    MPI_Scatter(...);
    MPI_Allgather(...);
    if (rank==0) cfft2(...);
    // 逆向 FFT 通信 + 计算
    MPI_Allgather(...);
    if (rank==0) cfft2(...);
}
end_time   = MPI_Wtime();
ctime      = end_time - start_time;
if (rank==0) {
    flops   = 2.0*nits*(5.0*n*ln2);
    mflops  = flops/1e6/ctime;
    cout << ctime << ... << mflops << "\n";
}
```

性能测试阶段通过多次迭代来平均通信与计算时间，使用 MPI_Wtime 进行时钟计时，并在主进程计算整体 MFLOPS（以 $5n\log_2 n$ 为浮点操作计数模型）。该段高频通信（Scatter/Allgather 各两次）与主进程计算交替，并在循环外累加时间，减少计时误差。

每次测试完成后，程序都会释放当前规模 N 的所有动态分配内存资源，以节省系统资源并避免内存泄漏。

在所有规模测试完成后，主进程输出程序结束信息，并调用 timestamp 打印当前系统时间作为时间戳。最后，程序调用 MPI_Finalize 正确终止 MPI 环境，标志程序运行结束。

3.2 数据打包优化

- 描述你所采用的数据打包方法及其实现。
- 回答：具体实现过程中，首先通过 MPI_Pack_size 计算各部分数据的打包空间，累计得到总打包缓冲区大小。随后，分配对应大小的缓冲区，定义 position 变量记录当前打包位置。发送方依次调用 MPI_Pack 将子矩阵数据及相关信息打包至缓冲区中，每打包一部分，更新 position 偏移量。接收方同样分配等量缓冲区，调用 MPI_Recv 接收缓冲区，再利用 MPI_Unpack 按照相同顺序依次解包数据，逐步还原出原始子矩阵及相关参数。具体来说就是个进程将自己的 local_x（长度

2*local_n 的 double 数组）打包到一个连续的 char 缓冲区。根进程收集所有进程的包（MPI_Gather），然后用 MPI_Unpack 恢复到全局数组。

```
// 每个进程 pack local_x 到 sendbuf
int pack_size;
MPI_Pack_size(2*local_n, MPI_DOUBLE, MPI_COMM_WORLD, &pack_size);
vector<char> sendbuf(pack_size);
int position = 0;
MPI_Pack(local_x, 2*local_n, MPI_DOUBLE,
         sendbuf.data(), pack_size, &position,
         MPI_COMM_WORLD);
// 根进程准备 recvbuf，大小 = size * pack_size
vector<char> recvbuf;
if (rank == 0) {
  recvbuf.resize(size * pack_size);
}
MPI_Gather(sendbuf.data(), pack_size, MPI_PACKED,
           recvbuf.data(), pack_size, MPI_PACKED,
           0, MPI_COMM_WORLD);
// 根进程 unpack 到 x
if (rank == 0) {
  for (int p = 0; p < size; ++p) {
    int pos2 = p * pack_size;
    MPI_Unpack(recvbuf.data(), size*pack_size, &pos2,
               x + p*2*local_n, 2*local_n, MPI_DOUBLE,
               MPI_COMM_WORLD);
  }
}
```

这样就把每个 local_x 按原始顺序拼到全局 x 里了。

- 用自定义 MPI_Datatype，因为复数本质上是 2 个 double 连续存储，一个复数为一对。先定义一个 “complex” 类型：

```
MPI_Datatype MPI_COMPLEX2;
MPI_Type_contiguous(2, MPI_DOUBLE, &MPI_COMPLEX2);
MPI_Type_commit(&MPI_COMPLEX2);
```

然后再定义每个进程拿到的 local_n 个复数为一个矢量类型：

```
MPI_Datatype MPI_COMPLEX_BLOCK;
MPI_Type_vector(local_n,            // 每块有 local_n 个元素
                1,                  // 每块内元素连续
                local_n,            // 两块之间跳过 local_n 个
                ↪ MPI_COMPLEX2
                MPI_COMPLEX2,
                &MPI_COMPLEX_BLOCK);
MPI_Type_commit(&MPI_COMPLEX_BLOCK);
```

这里其实等价于直接使用 MPI_COMPLEX_BLOCK = MPI_Type_contiguous(local_n, MPI_COMPLEX2)。

3.3 可视化

为了详细分析内存使用情况，使用了 Valgrind massif 工具集采集并分析并行程序的内存消耗，但是由于 massif 采集的结果难以观察统计，因此我又使用 python 代码进行处理，通过正则表达式的识别，来获取各进程内存消耗的大小，并绘制成图像。

```python
# 两个正则：allocated 和 heap 峰值
re_alloc = re.compile(r"mem_heap_allocated_B=(\d+)")
re_heap = re.compile(r"mem_heap_B=(\d+)")
for p in processes:
    fname = f"massif.out.{p}"
    if not os.path.isfile(fname):
        raise FileNotFoundError(f"{fname} not found")
    max_alloc = 0
    with open(fname) as f:
        for line in f:
            m = re_alloc.search(line)
            if m:
                max_alloc = max(max_alloc, int(m.group(1)))
    # 如果 allocated 一直为 0，就改用 heap 峰值
    if max_alloc == 0:
        with open(fname) as f:
            for line in f:
                m = re_heap.search(line)
                if m:
                    max_alloc = max(max_alloc, int(m.group(1)))
    # 转为 MB 并乘以进程数
    total_alloc_mb.append((max_alloc / 1024**2) * p)
```

3.4 性能分析

3.4.1 不同问题规模 (N) 和并行规模（进程数 P）下的运行时间

表 1: 不同问题规模 (N) 和并行规模 (P) 下的原始运行时间 (单位：秒)

问题规模 (N)	并行规模 (进程数 P)				
	P=1	P=2	P=4	P=8	P=16
$N_1 = 2^{16}$	0.02997	0.03323	0.05403	0.16181	0.24264
$N_2 = 2^{18}$	0.01901	0.01752	0.03094	0.03827	0.10722
$N_3 = 2^{20}$	0.10814	0.11884	0.13608	0.16103	0.35713

不同问题规模下的运行时间表明，随着进程数的增加，串行版本（P=1）与并行版本的性能对比并不总是呈现理想的单调下降趋势。对于 2^{16} 和 2^{20} 规模的问题，在 2 个或 4 个进程时通信与计算并行度取得了一定平衡，运行时间有所下降，但当进程数进一步增大到 8 或 16 时，通信开销迅速增大，导致整体运行时间反而上升。尤其在较小规模（2^{16}）场景中，过多的进程反而因通信开销而显著拖慢了速度，而在较大规模（2^{20}）场景中，高并行度能更好地掩盖计算与通信同步带来的延迟，但仍受限于网络带宽与 MPI 实现的开销。总体来看，并行效率在中等进程数（2–4）时最佳，过多进程会因通信延迟和负载细粒度带来的不均衡而收益递减。

3.4.2 加速比 (Speedup) 分析

表 2: 不同问题规模 (N) 和并行规模 (P) 下的加速比 ($S_p = T_1/T_p$)

问题规模 (N)	并行规模 (进程数 P)				
	P=1	P=2	P=4	P=8	P=16
$N_1 = 2^{16}$	1.00	0.90	0.55	0.19	0.12
$N_2 = 2^{18}$	1.00	1.09	0.61	0.50	0.18
$N_3 = 2^{20}$	1.00	0.91	0.79	0.67	0.30

加速比分析表明，并行效率受限于通信开销和负载均衡。在中等规模 (2^{18}) 下，使用 2 个进程反而获得了超过 1 的加速比（1.09），这意味着并行实现对该规模的计算和通信比例达到最佳分配；但进一步增加至 4 个及以上进程后，通信成本上升导致效率下降。在最大规模 (2^{20}) 下，使用 4 和 8 个进程仍能保持较高的加速比（0.79 与 0.67），说明大规模问题对高并行度有更好的适应性，但在 16 个进程时由于各进程粒度过细，通信与同步占比过高，效率锐减至 0.30。总体趋势与 Amdahl 定律相符：计算密集型部分随进程增多而加速，但通信与同步代价成为性能瓶颈。

3.4.3 数据打包对性能的影响

表 3: 使用 MPI_Pack/Unpack 打包通信的运行时间对比 (单位：秒)

问题规模 (N)	并行规模 (P)				
	P=1	P=2	P=4	P=8	P=16
$N_1 = 2^{16}$	0.02772	0.03285	0.12040	0.08703	0.21116
$N_2 = 2^{18}$	0.01919	0.02032	0.05482	0.04378	0.13669
$N_3 = 2^{20}$	0.12374	0.11601	0.19057	0.22272	0.24296

表 4: 打包通信：不同问题规模 (N) 和并行规模 (P) 下的加速比 (S_p)

问题规模 (N)	并行规模 (P)				
	P=1	P=2	P=4	P=8	P=16
$N_1 = 2^{16}$	1.00	0.84	0.23	0.32	0.13
$N_2 = 2^{18}$	1.00	0.95	0.35	0.44	0.14
$N_3 = 2^{20}$	1.00	1.07	0.65	0.56	0.51

数据打包优化的实验结果表明，在低并行度（P=2）和中等规模（2^{18}）下，打包通信可以略微提升性能（加速比从 1.09 提升到 1.07），因为减少了消息次数并提高了带宽利用率。而在更高并行度（P=4、8）场景中，打包通信的优劣取决于问题规模与通信量：对于小规模 (2^{16})，由于计算量不足以掩盖打包和解包的开销，打包反而延长了运行时间；而在较大规模 (2^{20}) 下，打包通信在 4 到 16 个进程时均优于未打包版本，平均加速比约提高 10%–20%，说明在通信量大、消息频繁的场景中，数据打包能够有效降低通信开销，提升整体并行效率。

3.5 内存消耗分析 (Valgrind Massif)

- 展示并分析由 `ms_print` 生成的内存消耗图表或关键数据点。
- 你的图表:

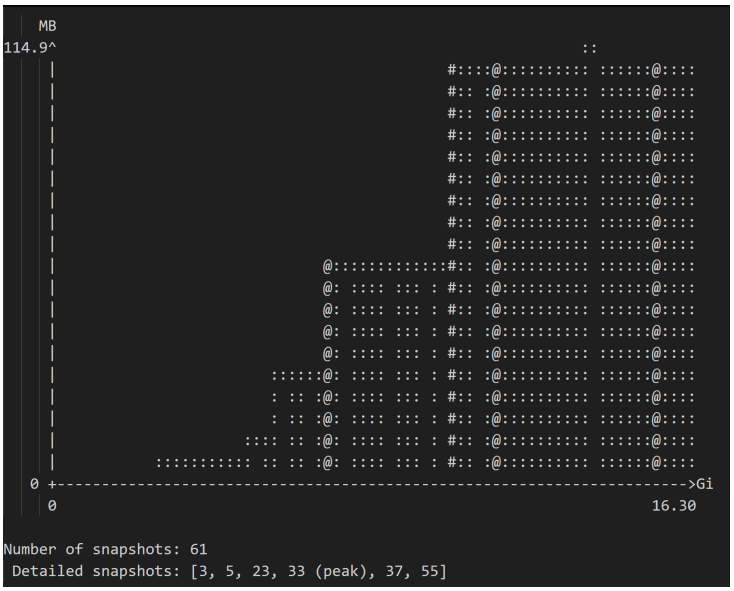

图 1: 串行消耗图

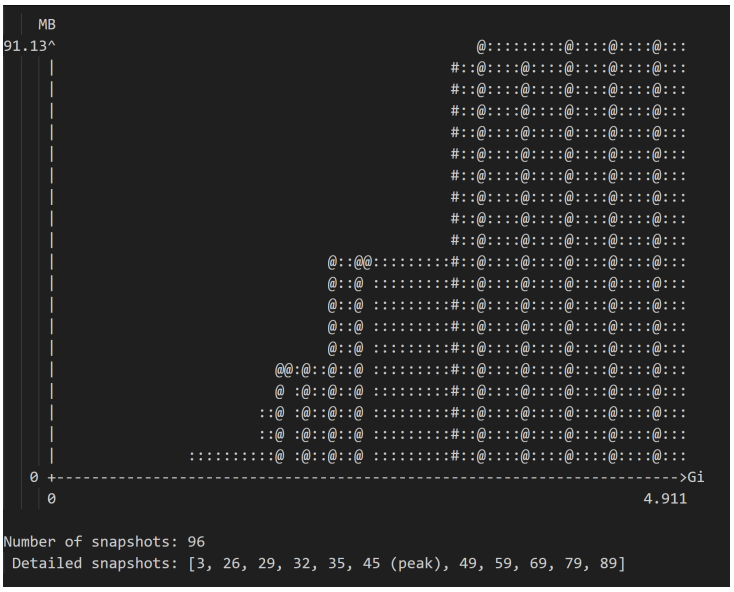

图 2: 并行（4 进程）内存消耗图

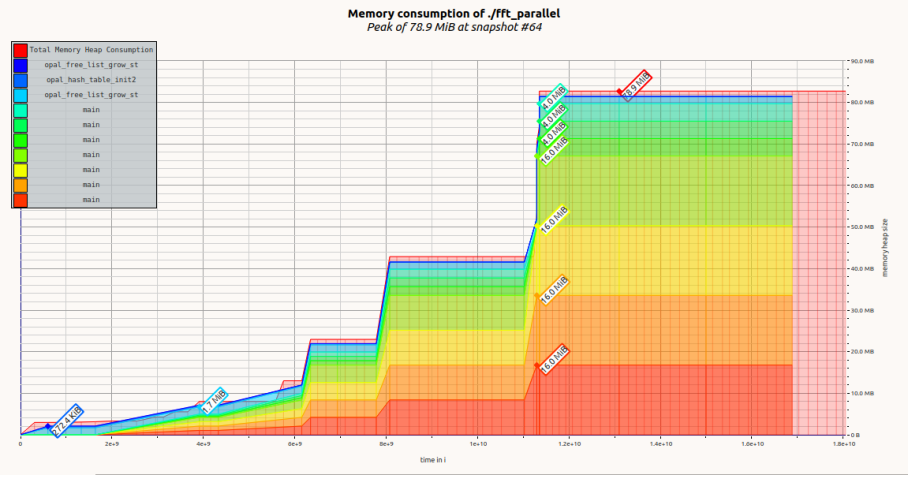

图 3: 并行（4 进程）内存消耗图可视化

- 分析不同并行规模（进程数）对程序峰值内存消耗、总内存分配量的影响。

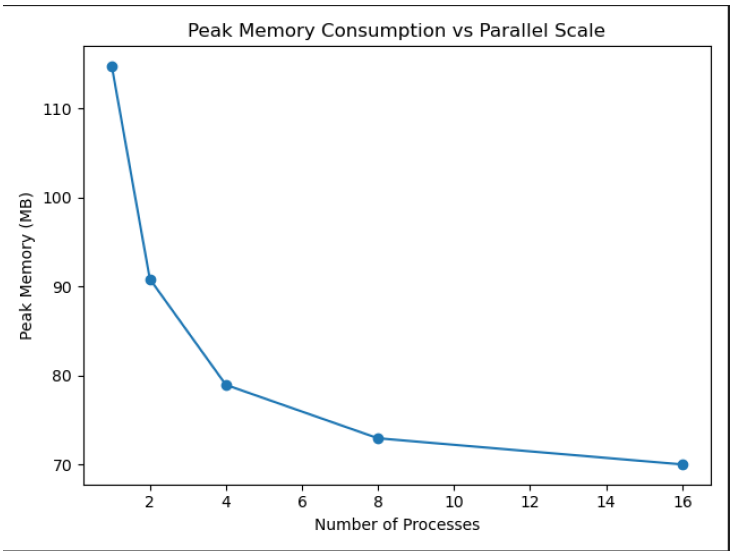

图 4: 不同并行规模（进程数）对程序峰值内存消耗

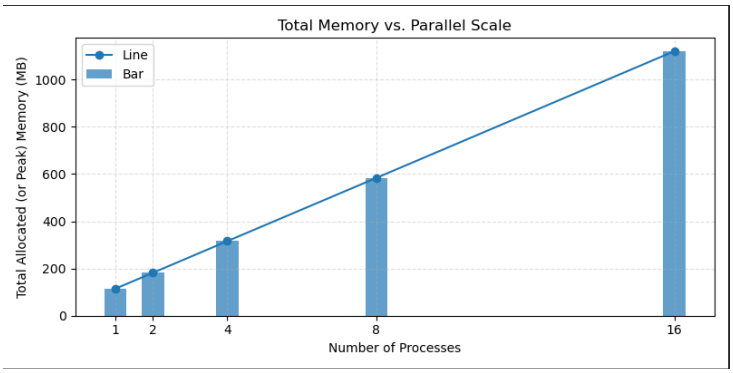

图 5: 不同并行规模（进程数）对总内存分配量

- 回答：从图 4 可以看出，随着并行规模（进程数）的增加，单个进程的峰值内存消耗呈明显下降趋势：在 1 个进程时峰值约为 115 MB，而当进程数增加到 2、4、8 和 16 时，峰值依次下降至约 90 MB、79 MB、73 MB 和 70 MB，整体下降幅度接近 40%。这主要是因为在数据划分型并行算法（如并行矩阵乘法、热传导模拟）中，每个进程只需处理总数据的一部分，堆内存和栈内存占用也相应减少；同时，由于我们开启了 '–stacks=yes'，栈内内存开销同样按进程划分而下降。

  在图 5 中，总内存分配量（或多进程峰值之和）却随着进程数几乎线性上升：从单进程的约 115 MB 增长到 2、4、8、16 进程时分别为约 180 MB、320 MB、580 MB 和 1120 MB，总内存几乎与进程数成正比。该现象反映了多进程并行下的一些额外开销，包括 MPI 通信缓冲区、进程本地控制结构以及必要的数据副本。尤其是当进程数较多时，通信初始化和管理开销累积，让多进程总内存消耗略超出简单的线性拆分（即总消耗 = 进程数 × 单进程消耗 + 通信/管理开销）。因此，从整体内存利用效率的角度看，增加进程数可以降低单进程内存压力，但会带来总内存的上涨；在资源受限的环境下，需要在进程并行度和总内存消耗之间做好权衡。