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 李源卿

学 号 22336128

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_str` 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

- 描述你的并行 FFT 算法设计和实现的关键点。

3.1.1 并行思路

如下图所示，如果有两个进程，我会将系数组成的数组 x 分拆为两个 local_x，然后进程先计算 (m-log2) 轮，之后由进程 0 来收集数据并进行后续的计算。那么，假设进程数为 p，那么我们就能并行计算 (m-logp) 轮，再进行串行汇总。从该角度出发，我的并行改编是弱可拓展性的，因为总有 logp 轮无法被并行化。

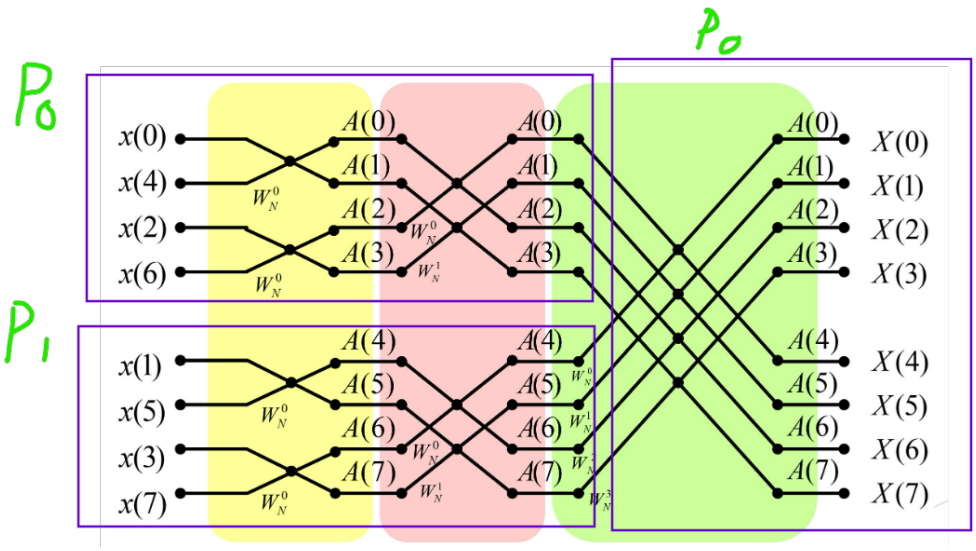

图 1: 进程数为 2

3.1.2 代码说明

（都在注释里了）

```
// 并行FFT计算部分
// 计算每个进程处理的复数数量（每个复数含实部和虚部）
int local_n = n / size;
double* local_x = new double[2 * local_n];  // 本地输入数组
double* local_y = new double[2 * local_n];  // 本地输出数组
MPI_Barrier(MPI_COMM_WORLD);  // 同步所有进程
// 主进程分发数据到各进程
MPI_Scatter(para_x, 2*local_n, MPI_DOUBLE,
            local_x, 2*local_n, MPI_DOUBLE,
            0, MPI_COMM_WORLD);
```

```
double ptime1 = MPI_Wtime();  // 开始计时并行计算
// 执行本地FFT计算
int mj = cfft2(local_n, local_x, local_y, w, sgn);
// 收集各进程结果到主进程
MPI_Gather(local_y, 2*local_n, MPI_DOUBLE,
           para_y, 2*local_n, MPI_DOUBLE,
           0, MPI_COMM_WORLD);
// 主进程合并各进程结果
if(rank == 0) {
    int cnt = 0;
    bool tagl = 0;  // 用于切换输入/输出数组
    while (mj < n/2) {  // 逐步合并直到完成所有阶段
        cnt++;
        mj *= 2;  // 处理的数据块大小翻倍
        if(tagl) {
            // 使用para_x作为输入，para_y作为输出，来回倒
            step(n, mj, &para_x[0], &para_x[(n/2)*2],
                 &para_y[0], &para_y[mj*2], w, 1.0);
            tagl = 0;
        } else {
            // 使用para_y作为输入，para_x作为输出
            step(n, mj, &para_y[0], &para_y[(n/2)*2],
                 &para_x[0], &para_x[mj*2], w, 1.0);
            tagl = 1;
        }
    }
}
// 如果发现结果被倒在了x里，把结果copy到y中
if(rank == 0 ){
    if (tagl){
        ccopy(n, para_x, para_y);
    }
}
```

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。

3.2.1 复数类型

```
MPI_Datatype MPI_Complex;
int blocklens[2] = {1, 1};
MPI_Aint displs[2] = {0, sizeof(double)};
MPI_Datatype types[2] = {MPI_DOUBLE, MPI_DOUBLE};
MPI_Type_create_struct(2, blocklens, displs, types, &MPI_Complex);
MPI_Type_commit(&MPI_Complex);
```

将两个双精度浮点数打包为一个 MPI_Complex，如上所示

3.2.2 w 数组打包

```
MPI_Datatype MPI_W_FullArray;
MPI_Type_contiguous(n/2, MPI_Complex, &MPI_W_FullArray);
MPI_Type_commit(&MPI_W_FullArray);
MPI_Bcast(w, 1, MPI_W_FullArray, 0, MPI_COMM_WORLD);
```

将 w 数组打包后进行广播。

3.3 性能分析

- **不同问题规模 (N) 和并行规模（进程数 P）下的运行时间：**
  - 请以表格形式展示在不同 N 和 P 组合下的原始运行时间。请根据你的实际实验情况填写数据。

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

问题规模 (N)	并行规模 (进程数 P)				
	P=1 (串行)	P=2	P=4	P=8	P=16
$N_1 = 2^{16}$	0.011631	0.00653672	0.0039092	0.0046048	0.00500577
$N_2 = 2^{18}$	0.0486345	0.0294647	0.0170588	0.0190368	0.0182605
$N_3 = 2^{20}$	0.229325	0.133955	0.0832629	0.100612	0.117159

- **加速比 (Speedup) 分析：**
- 分析加速比，讨论其是否符合预期，并解释原因（例如，通信开销、负载均衡等）。
  - 加速比是符合预期的，因为除了并行计算 fft 的耗时，还有集合通信和串行收集结果＋计算剩余 fft 部分的耗时，大部分加速比没有超过 3 是在预期之内的。

表 2: 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 (进程数 P)				
	P=1 (串行)	P=2	P=4	P=8	P=16
$N_1 = 2^{16}$	1.00	1.78	2.98	2.53	2.32
$N_2 = 2^{18}$	1.00	1.65	2.85	2.56	2.66
$N_3 = 2^{20}$	1.00	1.71	2.75	2.28	1.96

  - 当进程数目超过四之后，随着进程数的增多，串行部分的计算轮数也会变多，所以性能下降（加速比变小，时间变长）是在预期之内的。
  - 进程数小于四时，虽然增多进程数会导致串行部分增多，但是显然并行计算 fft 带来的收益更大。

- **数据打包对性能的影响：**
  - 分析数据打包带来的性能提升或可能引入的额外开销。
    * 由于我的并行程序的设计方法不涉及特别多的集合通信操作，所以数据打包带来的优化有限，所以我们可以看到其性能上没有大的变化。

表 3: 数据打包下不同问题规模 (N) 和并行规模 (P) 下的运行时间 (单位：秒)

问题规模 (N)	并行规模 (进程数 P)				
	P=1 (串行)	P=2	P=4	P=8	P=16
$N_1 = 2^{16}$	0.0119874	0.00633955	0.00434184	0.00511098	0.00463176
$N_2 = 2^{18}$	0.0501239	0.0290866	0.0162928	0.0184169	0.019194
$N_3 = 2^{20}$	0.232296	0.130888	0.0826974	0.130924	0.161699

表 4: 数据打包下不同问题规模 (N) 和并行规模 (P) 的加速比 (S_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	1.89	2.76	2.35	2.59
$N_2 = 2^{18}$	1.00	1.72	3.08	2.72	2.61
$N_3 = 2^{20}$	1.00	1.77	2.81	1.77	1.43

3.4 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。

- **关键数据点：**
  - **MPI 初始化阶段**（snapshot=6 到 snapshot=20）：内存从 198KB 增长到 265KB，主要由 MPI 进程通信缓冲区分配驱动
  - **并行计算阶段**（snapshot=20 到 snapshot=64）：内存呈阶梯式增长，每次增长约 200-500KB，对应 FFT 计算中的 MPI_Scatter/MPI_Gather 操作
  - **内存释放阶段**（snapshot=70 之后）：内存从 8.2MB 骤降至 257KB，显示程序在结束时正确释放了 MPI 通信缓冲区
- **内存模式特点：**
  - MPI 进程间通信（特别是 Scatter/Gather）是主要内存消耗源
  - 未发现明显内存泄漏，程序结束时的残余内存仅 87KB（snapshot=74）
- 你的图表：（进程数为 4）

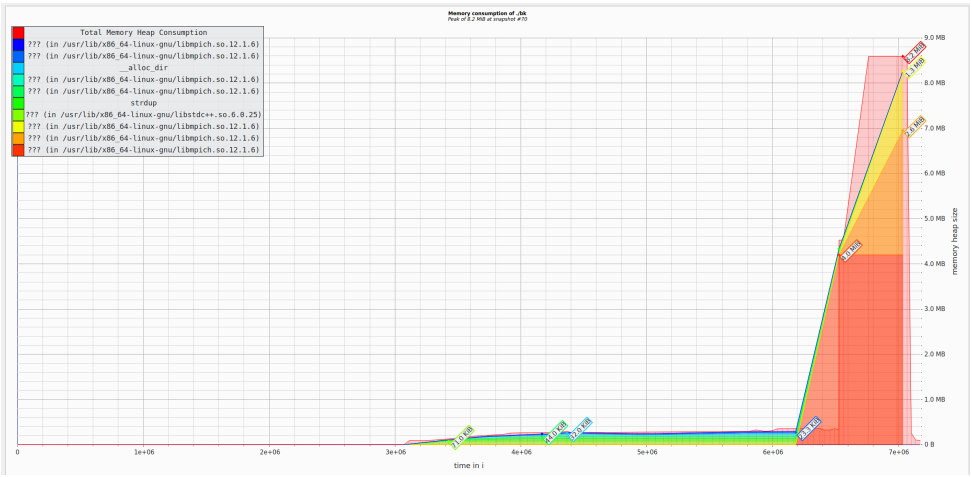

图 2: 不进行集合通信（P=4）

- 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。
- 回答：以进行集合通信为例子：
  - 进程数对内存消耗的峰值影响不大，峰值消耗都为 8.2 MiB，堆上的开销为 34.9 KiB.
  - 所有进程的主要内存分配均来自以下来源：C++ 标准库 （libstdc++.so.6）：初始化和动态链接过程的内存占用；MPICH 库 （libmpich.so.12.1.6）：可能是 strdup、PMPI_Init 等 MPI 初始化相关操作.
  - 堆树结构相似 ：所有文档的 heap_tree=detailed 部分均包含类似的分层结构，说明内存分配路径在不同进程数下保持一致（除了 local 数组，其他的我在程序开始时就已经分配好了）。

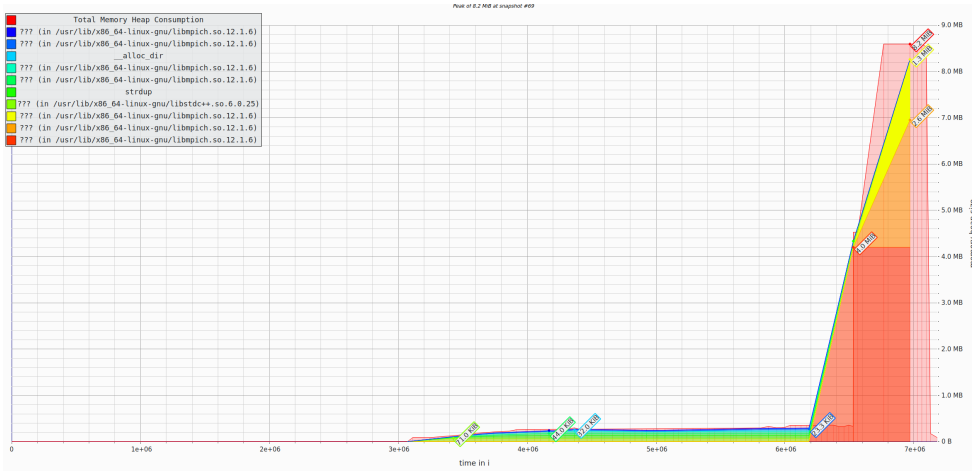

图 3: 进行集合通信（P=4）

– 峰值内存均出现在程序后期，可能与 local 数组的申请以及 MPI 的集合通信有关。