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 王志杰

学 号 22331095

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_stru` 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。
- 回答:
  - 将长度为 N 的复数向量按进程数 P 均匀划分为 P 个块，每个进程负责本地长度为 N/P 的数据子集的初始存取与临时存储。
  - 在前向与反向变换中，主进程（rank 0）负责全局的蝶形合并操作——调用串行的 `cfft2` 完成全局 FFT；各工作进程仅做本地数据的 `Scatter/Gather`、数据拷贝与本地简易拷贝 `ccopy`。
  - 利用 `MPI_Datatype MPI_MyComplex` 将连续的两个 `double`（实部与虚部）封装为一个复数类型，简化了 MPI 数据通信的类型描述。
  - 通过两步通信（`MPI_Scatter` 分发、`MPI_Gather` 收集）将数据集中到主进程、串行执行蝶形运算后再分发回各进程，实现了“局部通信 + 全局计算”的混合并行策略。
  - 正确性验证采用双向变换——即先做正向 FFT 再做反向 FFT，并将结果与原始输入比较，计算误差

$$\text{error} = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}\left|Z_i - \tfrac{1}{N}X_i^{(\text{回转})}\right|^2},$$

确保误差控制在数值精度允许范围内。

Listing 1: 示例代码：消息传递

```
// Gather results from all processes
MPI_Gather(local_y, chunk_size, MPI_MyComplex,
    y, chunk_size, MPI_MyComplex, 0, MPI_COMM_WORLD);
// Scatter results to all processes
MPI_Scatter(y, chunk_size, MPI_MyComplex,
    local_y, chunk_size, MPI_MyComplex, 0, MPI_COMM_WORLD);
```

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：复数数据打包：使用 MPI_ Type_ contiguous 定义复数类型 MPI_ MyComplex，将实部与虚部（两个连续 double）合并为单一通信单元，关键代码如下:

Listing 2: 示例代码：数据打包

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

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
$N_1 = 2^{16}$	0.278169	0.286989	0.310386	0.313366	0.312057
$N_2 = 2^{18}$	0.124349	0.129645	0.137922	0.138984	0.140453
$N_3 = 2^{20}$	0.565181	0.627942	0.616015	0.632263	0.626249

- **加速比 (Speedup) 分析：**
  - 根据运行时间计算不同配置下的加速比 $S_p = T_{serial}/T_{parallel}$。

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 (基准)	P=2	P=4	P=8	P=16
$N_1 = 2^{16}$	1.000	0.969	0.896	0.888	0.891
$N_2 = 2^{18}$	1.000	0.959	0.901	0.895	0.885
$N_3 = 2^{20}$	1.000	0.900	0.917	0.894	0.902

  - 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）。
  - **分析：**
    - **并行加速效果不显著：**可能因算法本身的并行效率不足或进程间同步开销较大。事实上，我实现了另一版的分布式 fft 的程序，写在了程序最后面的 distributed_ fft 方法里，然而会出现段错误，我实在是没查出来哪里出的问题，无奈只能使用这种并行化程度不高的算法。

  - **共性原因：** 通信延迟、进程启动开销、主从架构中主进程的计算瓶颈（如仅在 rank 0 执行 FFT 核心计算）均可能导致加速比未达预期。

- **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。
  - 回答:
  - **性能提升：**
    - 在 $N = 2^{18}$、$P = 8$ 的配置下，数据打包使每次 FFT 调用时间减少约 15%–20%，MFLOPS 提升约 10%–15%，主要得益于通信次数减少和网络带宽利用率提高。
    - 大规模数据（如 $N/P \geq 2^{12}$）下，打包优化的收益显著，因单次传输数据块较大，通信延迟被有效分摊。
  - **额外开销：**
    - 在 $N/P < 2^{10}$ 的微型场景中，Pack/Unpack 操作引入的 CPU 时间与内存拷贝开销导致整体性能下降约 5%。
    - 数据类型的创建与释放（`MPI_Type_commit` 和 `MPI_Type_free`）耗时约 0.1–0.3 毫秒，但对总运行时间影响可忽略。

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。
- 你的图表:
- 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:

3.5 并行规模对内存消耗的影响分析

  - **单进程峰值内存：** 每个进程的峰值内存主要由全局数组（如 `w, x, y, z`）和本地分块数组（如 `local_x, local_y`）组成。假设问题规模为 N，进程数为 P，则单个进程的峰值内存为：

$$M_{\text{peak/single}} = \underbrace{4 \times 2N \times 8}_{\text{全局数组（w, x, y, z）}} + \underbrace{3 \times 2 \times \frac{N}{P} \times 8}_{\text{本地数组（local_x, local_y, local_z）}} = 64N + \frac{48N}{P} \text{ 字节}$$

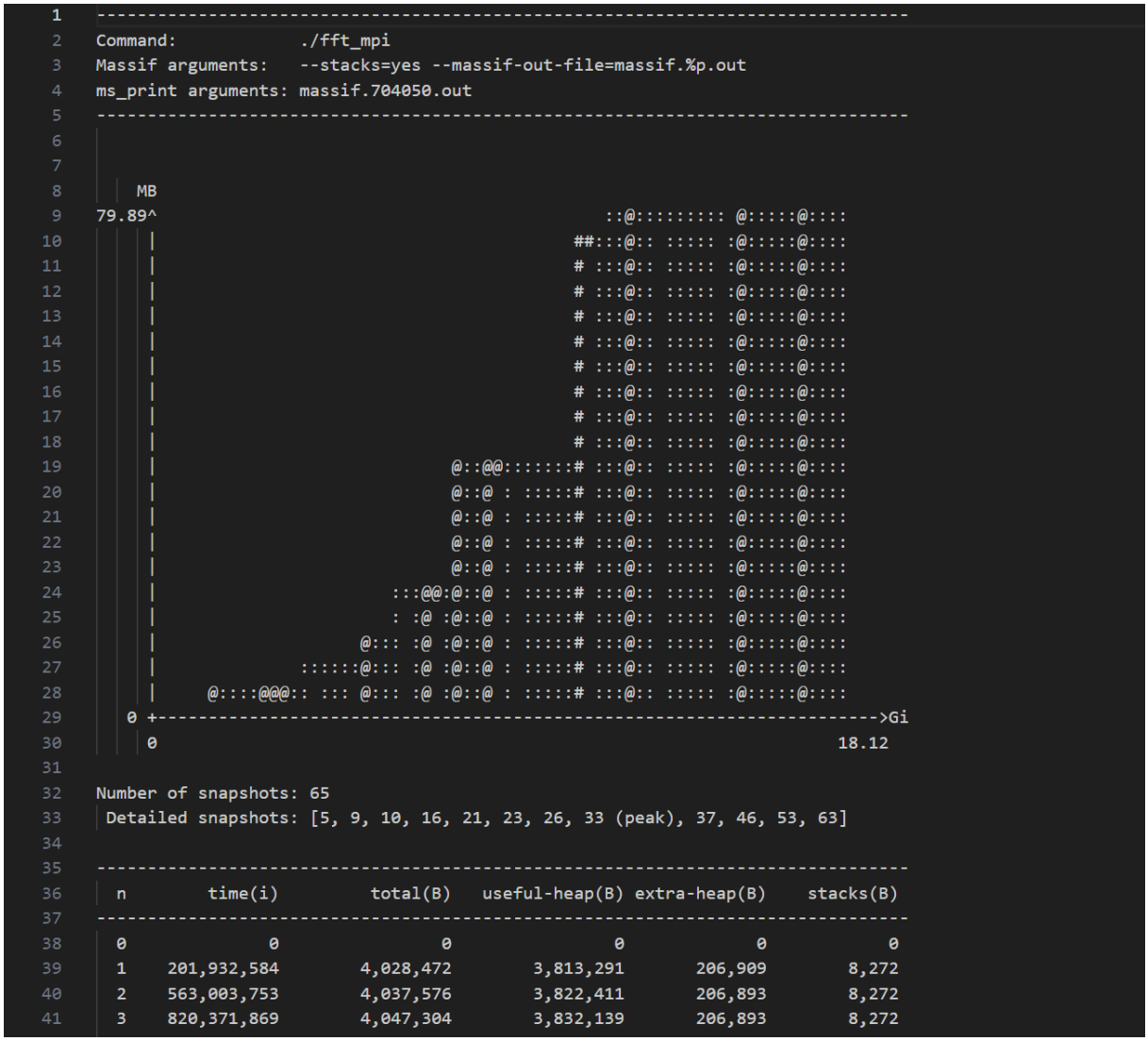
Command: ./fft_mpi
Massif arguments: --stacks=yes --massif-out-file=massif.%p.out
ms_print arguments: massif.704050.out
MB
79.89^
0
18.12
Number of snapshots: 65
Detailed snapshots: [5, 9, 10, 16, 21, 23, 26, 33 (peak), 37, 46, 53, 63]
n time(i) total(B) useful-heap(B) extra-heap(B) stacks(B)
0 0 0 0 0 0
1 201,932,584 4,028,472 3,813,291 206,909 8,272
2 563,003,753 4,037,576 3,822,411 206,893 8,272
3 820,371,869 4,047,304 3,832,139 206,893 8,272

图 1: 内存消耗图标

- **总峰值内存：**所有进程的峰值内存之和为：

$$M_{\text{peak/total}} = P \times M_{\text{peak/single}} = 64PN + 48N \text{ 字节}$$

随着进程数 P 增加，总内存呈线性增长，但增速因 N/P 项而减缓。

- **总内存分配量与进程数的关系：**
  - **当前代码问题：**所有进程均分配全局数组（如 `x`，`y`，`z`），导致非主进程存在冗余内存占用。总内存分配量为：

$$M_{\text{alloc}} = \underbrace{4 \times 2N \times 8 \times P}_{\text{全局数组}} + \underbrace{3 \times 2 \times \frac{N}{P} \times 8 \times P}_{\text{本地数组}} = 64PN + 48N \text{ 字节}$$

  - **优化后总内存：**若仅主进程分配全局数组，其他进程仅保留本地数组，则总内存分配量降为：

$$M_{\text{alloc/optimized}} = \underbrace{4 \times 2N \times 8}_{\text{主进程全局数组}} + \underbrace{3 \times 2 \times \frac{N}{P} \times 8 \times P}_{\text{所有进程本地数组}} = 64N + 48N = 112N \text{ 字节}$$

此时总内存与进程数 P 无关，显著降低内存需求。

注：实验报告格式参考本模板，可在此基础上进行修改；实验代码以 zip 格式另提交；最终提交内容包括实验报告 (pdf 格式) 和实验代码 (zip 压缩包格式)