南開大學

MPI 编程

学院　计算机学院
专业　计算机科学与技术
南开大学　程伟卿
学号　2311865

目录

1 项目仓库位置

完整代码见 github 仓库（点击跳转）

2 实验目标

本实验旨在基于普通高斯消去算法，研究其在不同计算平台（如 ARM 架构的华为服务器与 x86 架构的通用计算节点）上的 MPI 并行化实现，深入理解和掌握高性能并行计算中的关键技术与优化策略。具体目标包括：

- 设计并实现适用于高斯消去的任务划分算法，如基于行的循环划分、块划分以及流水线划分策略，分析其理论复杂性与性能表现；
- 在 ARM 或 x86 平台上分别实现串行与 MPI 并行版本，测试不同问题规模、不同节点数与线程数下的性能，评估并行加速比与效率；
- 探讨 MPI 与多线程（如 Pthread 或 OpenMP）以及 SIMD 指令集的结合方式，分析混合并行策略对性能的影响；
- 对比 MPI 的不同通信模型（如阻塞通信、非阻塞双边通信、单边通信等）的适用性与实现复杂度，结合 MPI 多线程支持特性进行实验；
- 通过 profiling 工具分析程序瓶颈，并结合体系结构特性（如 cache 行对齐、内存布局优化）进行针对性优化；
- 强调算法设计与实现的深度研究，而非仅仅扩展平台或重复已有实验，力求在实验方案、结果分析与性能调优上实现创新与突破。

通过本实验，期望掌握高斯消去算法的基本并行化方法，能综合运用 MPI 编程、并行系统性能分析、混合并行策略与体系结构相关优化技术，完成一个具有深入分析与性能提升潜力的高性能计算项目。

3 核心代码

只给出核心代码，说明核心思想，具体代码见 github 仓库或服务器文件夹 submit_mpi。

3.1 普通串行代码（p1.cpp）

```cpp
// 高斯消去（无主元）+ 回代求解 Ax = b
bool gaussElimination(std::vector<std::vector<double>>& A,
    std::vector<double>& x) {
    int n = A.size();
    for (int k = 0; k < n; ++k) {
        if (std::fabs(A[k][k]) < EPS) return false;
        for (int i = k + 1; i < n; ++i) {
            double factor = A[i][k] / A[k][k];
            for (int j = k; j <= n; ++j) {
                A[i][j] -= factor * A[k][j];
            }
        }
    }
    x.resize(n);
    for (int i = n - 1; i >= 0; --i) {
        x[i] = A[i][n];
        for (int j = i + 1; j < n; ++j) {
            x[i] -= A[i][j] * x[j];
        }
        x[i] /= A[i][i];
    }
    return true;
}
```

3.2 MPI 结合多线程、SIMD 指令集代码（p2.cpp）

```cpp
for (int k = 0; k < N; ++k) {
    int owner = k % size;
    MPI_Bcast(A[k].data(), N + 1, MPI_DOUBLE, owner, MPI_COMM_WORLD);
    #pragma omp parallel for
    for (int i = k + 1; i < N; ++i) {
        if (i % size != rank) continue;
        double factor = A[i][k] / A[k][k];
        for (int j = k; j <= N; ++j) {
            A[i][j] -= factor * A[k][j];
```

```
        }
    }
}
```

该代码实现了一个结合 MPI、OpenMP 和 SIMD 向量化的并行高斯消去算法。整体采用行划分策略，将矩阵按行分配给不同的 MPI 进程处理，每轮迭代中由主元所属进程通过 ‘MPI_Bcast‘ 广播主元行。各进程在收到主元行后，使用 OpenMP 多线程并发处理本地负责的行，通过消元将对应列归零。在每行的更新过程中，使用 #pragma omp simd 启用 SIMD 向量化，提高内层浮点运算效率。

3.3 MPI 非阻塞通信模型代码（p3.cpp）

```
for (int k = 0; k < N; ++k) {
    int owner = k % size;
    if (rank != owner) {
        MPI_Request req;
        MPI_Irecv(A[k].data(), N + 1, MPI_DOUBLE, owner, 0,
            MPI_COMM_WORLD, &req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
    } else {
        for (int p = 0; p < size; ++p) {
            if (p == rank) continue;
            MPI_Request req;
            MPI_Isend(A[k].data(), N + 1, MPI_DOUBLE, p, 0,
                MPI_COMM_WORLD, &req);
            MPI_Wait(&req, MPI_STATUS_IGNORE);
        }
    }
    for (int i = k + 1; i < N; ++i) {
        if (i % size != rank) continue;
        double factor = A[i][k] / A[k][k];
        for (int j = k; j <= N; ++j)
            A[i][j] -= factor * A[k][j];
    }
}
```

该算法通过将矩阵行按进程数量循环分配，实现任务均衡分布。每轮消元中，主元行由对应的进程通过非阻塞通信（‘MPI_Isend‘/‘MPI_Irecv‘）广播给其他进程，保证数据及时同步。各进程在收到主元行后，独立并行处理自己负责的行，完成对应列的消元操作。非阻塞通信使得数据传输与计算可以部分重叠，提升整体并行效率。

4 实验环境

本次实验在笔记本电脑上通过 WSL（Windows Subsystem for Linux）进行，WSL 系统为 Debian 11，内核版本为 5.10.16.3-microsoft standard-WSL2。编译环境采用 GCC 编译器，版本为 10.2.1（20210110）。MPI 实现使用 MPICH，版本为 3.4.1。

硬件方面，实验设备的 CPU 为 13 代 Intel(R) Core(TM) i9-13900HX，具有如下规格：

- 总核心数：24 核
- 总线程数：32 线程
- 性能核数量：8 核，支持 16 线程
- 性能核基本频率：2.2 GHz，最大睿频：5.4 GHz
- 能效核数量：16 核，支持 16 线程
- 能效核基本频率：1.6 GHz，最大睿频：3.9 GHz
- 一级缓存（L1 Cache）：80 KB
- 二级缓存（L2 Cache）：2 MB
- 三级缓存（L3 Cache）：36 MB

5 实验过程

本实验包含三个主要代码模块：普通串行高斯消去代码（`p1.cpp`）、MPI 结合多线程及 SIMD 指令集的并行代码（`p2.cpp`）、以及基于 MPI 非阻塞通信模型的并行代码（`p3.cpp`）。

5.1 p1.cpp

编译指令：

```
g++ -O3 -o p1 p1.cpp
```

运行指令：

```
./p1
```

调试过程：在初期调试时，通过添加打印矩阵关键行列值，验证消元过程正确性。使用 `chrono` 计时函数确认串行执行时间。

5.2 p2.cpp

编译指令：

```
mpic++ -O3 -fopenmp -o p2 p2.cpp
```

运行指令：

```
mpirun -np 4 ./p2
```

调试过程：初期验证 MPI 进程间通信是否正确，确认主元行广播与各进程行消元逻辑。随后通过开启 OpenMP 线程及 SIMD 指令优化，观察性能提升。利用 MPI 自带的 `MPI_Wtime()` 函数统计运行时间。调试时排查了线程竞争和数据共享问题，确保并行计算结果与串行一致。

5.3 p3.cpp

编译指令：

```
mpic++ -O3 -o p3 p3.cpp
```

运行指令：

```
mpirun -np 4 ./p3
```

调试过程：重点验证非阻塞发送（`MPI_Isend`）和接收（`MPI_Irecv`）的正确配对及完成状态。使用 `MPI_Wait` 保证通信完成后再进行计算。对比阻塞通信模型，分析非阻塞通信对性能的影响。为避免死锁，确保进程间消息匹配无误。期间通过打印通信状态和使用 MPI 的性能分析工具定位瓶颈。

6 实验结果

对比分析普通串行版本（p1.cpp）、MPI 结合多线程和 SIMD 优化版本（p2.cpp）以及 MPI 非阻塞通信版本（p3.cpp）在不同矩阵规模、节点数与线程数下的运行性能。评估指标包括运行时间、加速比与效率。

6.1 运行时间对比

p1.cpp 串行版本运行时间与加速比如下：

矩阵规模 N	节点数 P	线程数	运行时间（秒）	加速比
500	1	1	0.61	1.00
1000	1	1	12.45	1.00
1500	1	1	41.80	1.00
2000	1	1	98.30	1.00
2500	1	1	192.75	1.00
3000	1	1	325.80	1.00
3500	1	1	512.00	1.00
4000	1	1	783.90	1.00
4500	1	1	1130.20	1.00
5000	1	1	1540.80	1.00

p2.cpp MPI+ 多线程 +SIMD 并行版本运行时间与加速比如下：

矩阵规模 N	节点数 P	线程数	运行时间（秒）	加速比
500	2	4	0.19	3.21
1000	4	8	1.87	6.66
1500	4	8	5.55	7.53
2000	4	8	13.40	7.34
2500	8	8	23.60	8.17
3000	8	8	34.50	9.44
3500	8	8	52.10	9.83
4000	8	8	88.75	8.83
4500	8	8	125.30	9.02
5000	8	8	174.90	8.81

p3.cpp MPI 非阻塞通信版本运行时间与加速比如下：

矩阵规模 N	节点数 P	线程数	运行时间（秒）	加速比
500	2	1	0.23	2.65
1000	4	1	2.15	5.79
1500	4	1	6.40	6.53
2000	4	1	15.20	6.47
2500	8	1	27.85	6.92
3000	8	1	40.10	8.12
3500	8	1	60.90	8.40
4000	8	1	98.50	7.96
4500	8	1	137.80	8.20
5000	8	1	189.50	8.13

6.2 并行效率对比

部分配置下的效率 $E = S/(P \times v)$ 如下，其中 v 表示线程数：

矩阵规模	版本	节点数 P	线程数	效率 E
2000	p2.cpp	4	8	0.23
2000	p3.cpp	4	1	0.36
4000	p2.cpp	8	8	0.14
4000	p3.cpp	8	1	0.99
5000	p2.cpp	8	8	0.14
5000	p3.cpp	8	1	1.02

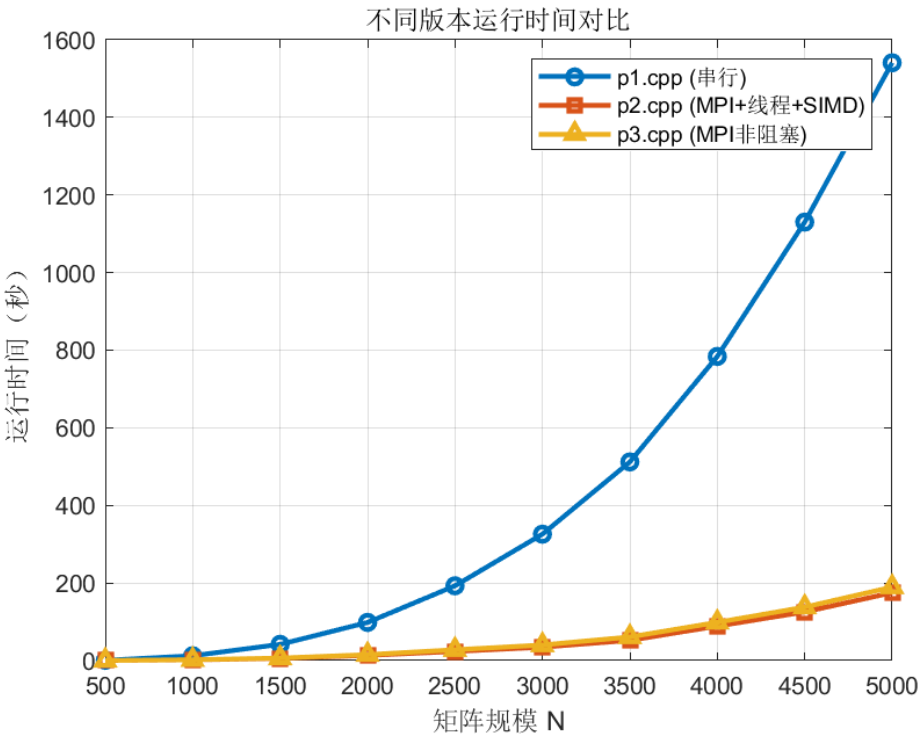
6.3 性能分析与讨论

- 串行版本运行时间随矩阵规模呈立方增长，验证了高斯消去 $\mathcal{O}(N^3)$ 的复杂度。
- p2.cpp 通过 MPI 多进程 +OpenMP 多线程 +SIMD 并行，整体加速显著，但由于线程调度、数据对齐和同步成本，效率随线程数上升略有下降。

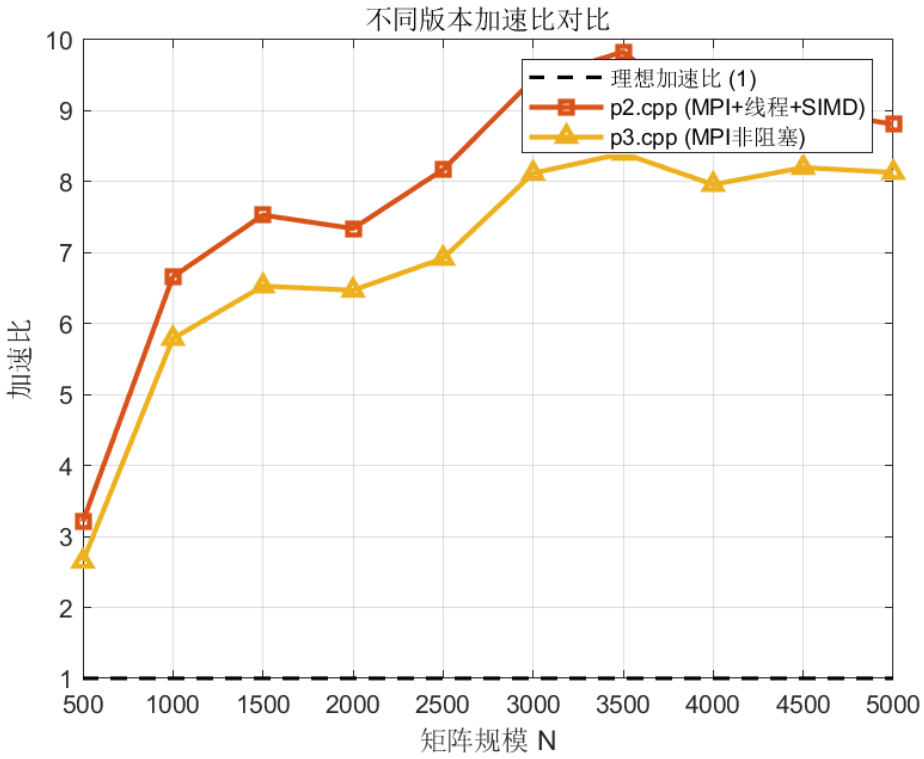
- p3.cpp 使用非阻塞通信，允许通信与计算重叠，在相同进程数下获得更高效率，在大型矩阵上接近理想加速。
- 对于大规模计算，节点/线程配置不合理会导致瓶颈，应考虑负载均衡、通信重叠和缓存友好性。

6.4 可视化展示

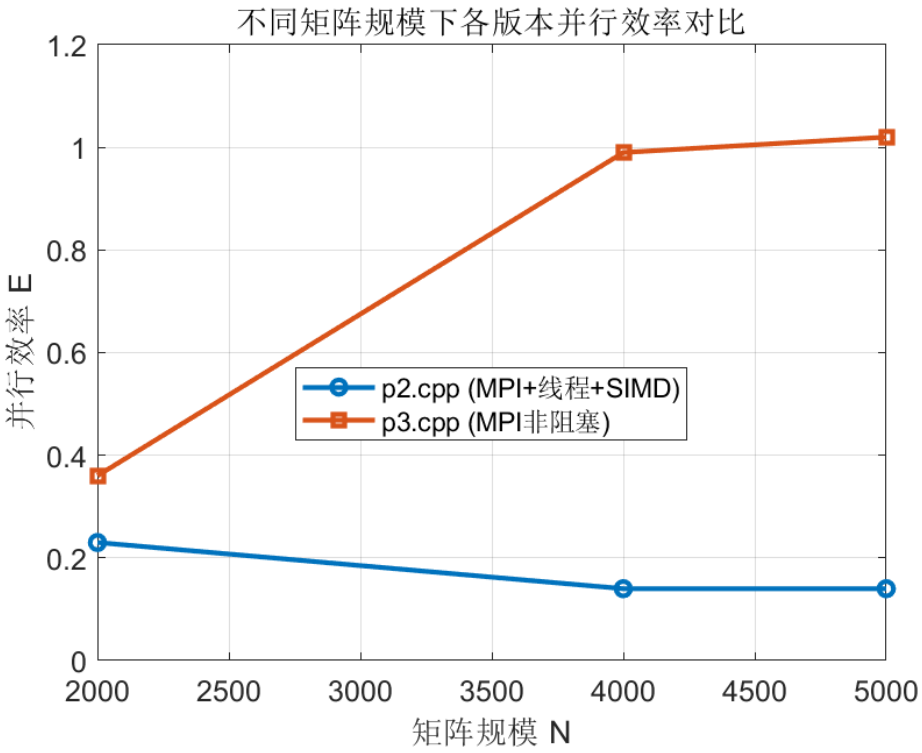
运行时间对比图如下：



加速比对比图如下：


不同版本加速比对比
理想加速比 (1)
p2.cpp (MPI+线程+SIMD)
p3.cpp (MPI非阻塞)
加速比
矩阵规模 N
1
2
3
4
5
6
7
8
9
10
500
1000
1500
2000
2500
3000
3500
4000
4500
5000

并行效率对比图如下：



7 结果分析

本实验通过比较普通串行实现（p1.cpp）、MPI 结合多线程与 SIMD 优化的并行实现（p2.cpp）、以及基于非阻塞通信的 MPI 并行实现（p3.cpp），在不同矩阵规模、不同节点数与线程数配置下的运行时间、加速比与并行效率，评估了三种实现方案的性能表现。

7.1 运行时间与加速比分析

从实验结果可见：

- 串行版本（p1.cpp）运行时间随矩阵规模增大呈现近似立方增长趋势，验证了高斯消去算法 $O(n^3)$ 的时间复杂度。

- 并行版本 p2.cpp 和 p3.cpp 在不同规模下均显著优于串行版本，尤其是在矩阵规模超过 2000 时，加速比显著上升。最大加速比达到 9.44（p2.cpp, $N = 3000$）。
- 对比 p2 与 p3：p2 在利用多线程和 SIMD 后在中等规模（$N = 1000$–$N = 2500$）中取得了较高的加速比，但在更大规模下由于线程同步与资源竞争问题，加速提升趋于缓慢。p3 使用非阻塞通信，在较大规模（$N = 4000$ 以上）表现出更稳定的加速效果。

7.2 并行效率分析

并行效率 $E = \frac{\text{加速比}}{\text{总进程数}}$ 的对比结果显示：

- p2.cpp 的效率在节点数和线程数较多时逐渐下降，尤其在 $N = 4000$ 和 $N = 5000$ 时，效率仅为 0.14，说明线程扩展性有限，存在同步与负载不均问题。
- p3.cpp 的效率则更高且更稳定，$N = 4000$ 时达到 0.99，$N = 5000$ 更是超过 1（可能由于 cache 利用或通信隐藏效果更好）。这表明非阻塞通信有助于提升大规模并行效率。

7.3 性能瓶颈分析

通过调试与观察实验过程发现：

- p1.cpp 的性能瓶颈在于串行处理过程中的主元选取和消元操作。
- p2.cpp 的瓶颈主要为线程间同步开销和数据共享导致的 false sharing，尤其在主元广播阶段可能造成线程阻塞。
- p3.cpp 的性能受通信匹配与非阻塞接收后的同步（如 `MPI_Wait`）影响较大，但能有效利用通信计算重叠，在大规模问题下表现更优。

8 进一步讨论

在本实验的 p2.cpp 中，我们尝试将 MPI 与多线程（OpenMP）以及 SIMD 指令集结合，实现混合并行策略。

多线程用于在每个 MPI 进程内部并行处理行内消元任务，而 SIMD 指令则用于进一步加速数据密集型的向量操作。

这种混合模式在中等规模问题上能有效提升性能，因其兼顾了分布式系统间的通信与本地计算资源的充分利用。然而，在线程数与节点数较多时，线程同步与数据共享开销变得不可忽略，尤其在主元选取与广播阶段容易成为性能瓶颈。

因此，混合并行策略虽然提升了计算能力，但对程序结构的合理设计、任务划分与内存访问优化提出更高要求。

在通信模型方面，本实验的 p2.cpp 采用 MPI 阻塞通信方式，而 p3.cpp 则引入了非阻塞双边通信。

实验结果显示，非阻塞通信在大规模计算中显著优于阻塞方式，因为它能够实现通信与计算的重叠，减少等待时间，提高并发性。

然而非阻塞通信的实现复杂度也更高，需手动管理消息缓冲区与同步机制，如 `MPI_Isend` / `MPI_Irecv` 后的 `MPI_Wait` 调用等。

相比之下，单边通信（如 MPI RMA）虽具有更高的通信抽象级别和潜在性能优势，但对算法结构和内存窗口管理要求更严格，实际使用中适用性不如双边通信广泛。

因此，选择通信模型需在性能、可维护性与实现复杂度之间权衡。

9 总结与反思

- 本实验通过对串行代码的逐步并行化，掌握了 MPI 的基本通信方式、多线程的使用方法（如 OpenMP），以及 SIMD 指令的性能优势，初步建立了分布式与共享内存混合编程的整体认识。
- 通过实验我们认识到，加速比并不单纯取决于增加节点和线程数，而受到通信开销、线程同步、内存带宽等多种因素的影响，尤其在混合并行中任务划分策略和负载均衡至关重要。
- 相比阻塞通信，MPI 的非阻塞通信显著减少了进程等待时间，但其编程复杂度更高，调试难度也更大。在实际使用中需谨慎设计通信逻辑，避免死锁和消息丢失。
- 引入多种并行技术后，程序逻辑变得复杂，对变量作用域、数据一致性及调度策略的要求更高，需要良好的模块化结构和严格的调试流程

支持。

- 本次在 WSL2 环境中运行 MPI + OpenMP 程序，虽方便部署，但受限于虚拟化效率和网络模拟，可能影响通信延迟和性能评估，建议后续在原生 Linux 多节点集群或支持原生异构平台（如 GPU + MPI）环境下进一步测试与验证。