MPI 编程实验
——以高斯消去为例

杜忱莹 周辰霏
2021 年 5 月
麦隽韵
2022 年 5 月
李君龙
2023 年 5 月
丁延峰
2024 年 5 月
华志远孔德嵘张逸非
2025 年 5 月

目录

1 实验介绍

1.1 实验选题

1. 基础选题：高斯消去法 (LU 分解)。
2. 进阶选题：ANN，NTT，口令猜测算法三选一。

1.2 实验要求

1. 基本要求（最高获得 90% 分数）：ARM 平台（华为服务器）上普通高斯消去计算的基础 MPI 并行化实验：
   - 设计实现适合的任务分配（数据划分）算法，分析其性能。
   - 在 ARM 平台上编程实现、进行实验，测试不同问题规模、不同节点数/线程数下的算法性能（串行和并行对比），讨论一些基本的算法/编程策略对性能的影响。
2. * 进阶要求（最高可获得 100% 分数）：ANN，NTT，口令猜测算法三选一，具体要求可参考各实验指导书。

2 多进程与多线程介绍

本次实验涉及到的 MPI 编程是多进程编程，什么是进程？本节介绍了线程和进程的区别于联系。首先明确：OpenMP 和 Pthread 均为多线程编程。

2.1 进程与线程

进程 (process) 是操作系统进行资源分配的最小单元，**线程 (thread)** 是操作系统进行运算调度的最小单元。

通常情况下，我们在运行程序时，比如在 Linux 下通过 *./test* 运行一个可执行文件，那么我们就相当于创建了一个进程，操作系统会为这个进程分配 ID 和堆栈空间等资源。如果这个可执行程序是个多线程程序（比如由 OpenMP 或 Pthread 编写的 cpp 文件编译而来），那么这个 *./test* 进程在执行到某个特定位置时会创建多个线程继续执行。所以，**进程可以包含多个线程，但每个线程只能属于一个进程**。

如图2.1所示，每个进程有独立的地址空间（外部框表示），同一进程内不同线程（曲线）共享该进程的内存。但是不同进程不共享地址空间，如果一个进程要访问另一个进程的数据就只能通过特定的函数进行通信。

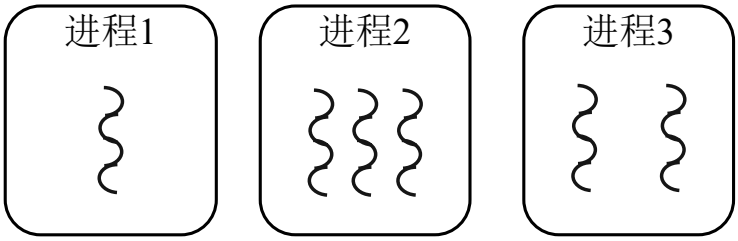

图 2.1: 进程与线程概念

2.2 MPI 多进程

区别于普通的可执行程序只创建一个进程,通过 $mpiexec -n\ num\ ./test$ 运行的程序会根据 $-n$ 选项后的参数 num，创建 num 个进程运行 $test$ 可执行程序。

课程中提到了在开始编写 MPI 部分代码时调用 MPI_Init 完成 MPI 初始化，在 MPI 部分结束时调用 MPI_Finalize 完成清理工作。这两个函数与多线程中创建和销毁线程有本质的区别：程序并非只在两个函数调用之间的部分并行执行，而是整体并行执行，在两个函数之间的部分完成数据划分运算、通信等工作。以下方程序为例，代码在两个函数调用的外部定义了 $size$ 变量和 $rank$ 变量，同时输出 $HelloWorld$!；在两个函数之间的部分为 $size$ 和 $rank$ 赋值并输出。编译并通过 $mpiexec\ -n\ num\ ./test$ 运行后，每个进程均会输出一次 $HelloWorld$!，共输出 num 次，这也验证并行并非只发生在 MPI_Init 和 MPI_Finalize 之间的代码。

```
1  #include"mpi.h"
2  #include<iostream>
3
4  int main(int argc, char* argv[]) {
5      int size,rank;
6      std::cout<<"Hello world!"<<std::endl;
7      MPI_Init(&argc, &argv);
8      size = MPI::COMM_WORLD.Get_size();
9      rank = MPI::COMM_WORLD.Get_rank();
10     std::cout<<"MPI size: "<<size<<" ,rank: "<<rank<<std::endl;
```

```
    MPI_Finalize();
    return 0;
}
```

3 高斯消去实验设计指导

3.1 实验总体思路

以高斯消去法为例：

1. 首先初始化生成矩阵元素值，按照之前作业给出的伪代码实现高斯消去法串行算法。

```
procedure LU (A)
begin
 for k := 1 to n do
   for j := k+1 to n do
     A[k, j] := A[k, j]/A[k, k];
   endfor;
   A[k, k] := 1.0;
   for i := k + 1 to n do
     for j := k + 1 to n do
       A[i, j] := A[i, j] − A[i, k] × A[k, j];
       endfor;
   A[i, k] := 0;
     endfor;
   endfor;
end LU
```

2. 使用课堂所学 MPI 编程设计实现合适的任务分配（数据划分）算法，按照不同任务划分方式（如块划分、循环划分等），分别设计并实现 MPI 算法。并考虑将其与 Pthread 算法以及 SIMD 算法结合。对各算法进行复杂度分析（基本的运行时间、加速比的分析以及更深入的伸缩性分析），思考能否继续改进。

以 MPI 按一维（行）块划分消去并行处理为例，假设设置 m 个 MPI 进程、问题规模为 n，则给每一个进程分配 n/m 行的数据，对于第 i 个进程，其分配的范围为 $[i*(n-n\%m)/m, i*(n-n\%m)/m+(n-n\%m)/m-1]$，而最后一个进程分配 $[(m-1)*(n-n\%m)/m, n-1]$ 行。注意，这种简单策略是将余数部分（$n\%m$ 行）都分配给了最后一个进程，大家可思考稍复杂些但负载更为均衡的分配策略——将余数部分均匀分配给前 $n\%m$ 个进程，

每个进程一行。初始化矩阵等工作由 0 号进程实现，然后将分配的各行发送给各进程。在第 k 轮消去步骤，负责第 k 行的进程进行除法运算，将除法结果一对多广播给编号更大的进程，然后这些进程进行消去运算。消除过程完成后，可将结果传回 0 号进程进行回代，也可由所有节点进行并行回代。

Algorithm 1: MPI 版本的普通高斯消元

Data: 系数矩阵 A[n,n], 进程号 myid, 负责的起始行 r1, 负责的终止行 r2, 进程总数 num

Result: 上三角矩阵 A[n,n]

```
1  for k = 0 to n-1 do
2      if r1 <= k <= r2 then
3          for j = k + 1; j < n; j++ do
4              A[k,j]=A[k,j]/A[k,k];
5          A[k,k] ← 1.0;
6          for j = 0; j < num; j++ do
7              MPI_Send(&A[k,0], n, MPI_FLOAT, j, ...);
8      else
9          MPI_Recv(&A[k,0], n, MPI_FLOAT, j, ...);
10     for i ← r1 to r2 do
11         for j = k + 1; j < n; j++ do
12             A[i,j] ← A[i,j]-A[k,j]*A[i,k];
13         A[i,k] ← 0;
14     k++;
15     ...
```

一维块循环划分略微复杂，主要是以更小任务粒度循环给各进程分配任务行号的计算会更复杂些，大家自行推导。一维列（循环）块划分与一维行划分略有不同，在除法阶段，需要持有对角线上元素的进程将其广播给其他进程，然后所有进程对自己所负责的列元素进行除法计算；接下来无需广播除法结果，因为需要除法结果的后续行都由同一个进程负责，直接在本地进行消去计算即可。这里就有一些可优化之处，大家可以考虑采用非阻塞通信、单边通信等手段降低对角线元素广播带来的进程空闲等待时间。

二维划分就更为复杂一些，大家需要更小心地计算每个进程负责的行号、列号，安排好通信，显然，采取二维划分后，既需要列方向广播对角线元素，也需要行方向广播除法结果。

流水线算法和普通的块划分的区别在于一个进程负责行的除法运算完成之后，并不是将除法结果一对多广播给所有后续进程，而是（点对点）转发给下一个进程；当一个进程接收到前一个进程转发过来的除法结果时，首先将其继续转发给下一个进程，然后再对自己所负责的行进行消去操作；当一个进程对第 k 行完成了第 $k-1$ 个消去步骤的消去运算之后，它即可对第 k 行进行第 k 个消去步骤的除法操作，然后将除法结果进行转发，如此重复下去，直至第 $n-1$ 个消去步骤完成。

3. 实验方面，改变矩阵的大小、进程数等参数，观测各算法运行时间的变化，对结果进行性能分析。测试相同并发度 (总线程数) 下不同节点数与每节点线程数的组合，并借助 Vtune profiling 等工具分析算法过程中的同步开销和空闲等待等。

3.2 MPI 的 C++ 编程

1. 头文件

所有进行 MPI 调用的程序单元必须包括 “mpi.h” 头文件。该文件定义了一些 MPI 常量，并提供了 MPI 函数原型。

```
#include "mpi.h"
```

2. MPI 预定义数据类型

```
MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_UNSIGNED_CHAR
MPI_FLOAT
MPI_DOUBLE
...
```

3. 常用的 MPI 函数

```
MPI_Comm_size//报告进程数
int MPI_Comm_size(MPI_Comm comm, int *size);

```

```
4  MPI_Comm_rank//报告识别调用进程的 rank，值从 0 size-1
5  int MPI_Comm_rank(MPI_Comm comm, int *rank);
6
7  MPI_Init//令 MPI 进行必要的初始化工作
8  int MPI_Init(
9      int*  argc_p /* 输入/输出参数 */,
10     char *** argv_p /* 输入/输出参数 */);
11
12 MPI_Finalize//告诉 MPI 程序已结束，进行清理工作
13 int MPI_Finalize(void);
14
15 MPI_Send//基本（阻塞）发，向一个进程发送数据
16 int MPI_Send(
17     void* buf  /* 存储数据的缓冲区地址 */,
18     int count  /* 发送的数据量 */,
19     MPI_Datatype datatype /* 数据类型 */,
20     int dest  /* 目的进程编号 */,
21     int tag  /* 标识向同一个目的进程发送的不同数据 */,
22     MPI_Comm /* MPI集群的通信域标识 */);
23
24 MPI_Recv//基本（阻塞）接收，从一个进程接收数据
25 int MPI_Recv(
26     void* buf  /* 存储数据的缓冲区地址 */,
27     int count  /* 接收的数据量 */,
28     MPI_Datatype datatype /* 数据类型 */,
29     int source  /* 源头进程编号 */,
30     int tag  /* 标识从同一个源头进程接收的不同数据 */,
31     MPI_Comm /* MPI集群的通信域标识 */,
32     MPI_Status *status /* 可以记录更多额外的信息 */);
```

更多更详细的内容请参考 MPI 的讲义。

3.3 作业注意要点及建议

1. 矩阵数值初始化问题

根据有些同学们反映，自己初始化矩阵在计算过程中会出现 inf 或 nan 的问题。这是由于精度问题以及非满秩矩阵造成的。inf 或 nan 的情况无疑会影响结果正确性的判断，也会在并行计算性能上造成一定影响，而随机生成数据的方式很明显无法保证能避免该问题尤其在规模巨大的情况下。个人建议初始化矩阵时可以首先初始化一个上三角矩阵，然后随机的抽取若干行去将它们相加减然后执行若干次，由于这些都是内部的线性组合，这样的初始数据可以保证进行高斯消去时矩阵不会有 inf 和 nan。

2. 不同任务划分策略

前面介绍了 MPI 任务分配（数据划分）策略，在此基础上，可结合多线程以及 SIMD 并行进行任务划分。例如，对于一维行划分，可看作是将第二层循环拆分，分配给不同进程；这样，继续进行多线程并行，即可继续对第二层循环进行划分，即，将进程负责的行划分给内部的多个线程，也可以对最内层循环进行划分，即，将进程负责的所有行的不同列分配给不同线程；而再继续结合 SIMD 并行化，则只能对最内层循环进行向量化。MPI 列划分、二维划分下与多线程和 SIMD 的结合类似，大家自己思考。

3. 计算误差与程序正确性

有关问题规模和并行计算由于重排了指令执行顺序和计算机浮点数所导致误差问题说明参考之前实验。

由于 n 较小的时候测出的计算时间也较小, 误差较大, 所以建议采取多次测量取均值的方法确定较合理的性能测试结果, 同时保证几种算法重复次数一致, 减少误差。而且考虑到实验平台等因素，计时测试工具可以考虑使用 MPI 计时。

4 ANN 选题指导

4.1 IVF

对于 IVF 算法，MPI 算法的实现思路和多线程基本一致，大家只需要将多线程的程序迁移到 MPI 上即可。另外大家也可以再额外探索一些内容，例如簇之间用 MPI 多进程并行，簇内再用多线程并行，同时距离计算使用 SIMD 加速。

4.2 图索引

对于图索引算法，可以尝试使用 IVF+HNSW，即先对数据集进行构建 IVF 索引，在 IVF 每个簇中建立 HNSW。在搜索时簇之间使用 MPI 并行搜索，如果实现了 HNSW 的查询内并行搜索，可以在簇内再利用多线程并行加速图索引的搜索。

另一个可以探索的算法是直接将数据集划分为 P（P 可以设置为 2-8，划分方式可以选择随机划分或者使用一些启发式的策略）个部分，然后对每个部分单独建立 HNSW，搜索时利用 MPI 在 HNSW 间进行并行搜索。

除了上述介绍的内容，同学们可以自由讨论其他 IVF 和 HNSW 的组合策略，基本思想都是对数据集做进行一些划分，然后在不同的划分间并行搜索。

5 NTT 选题指导

5.1 新的规约优化

目前多数同态加密库里的模数优化并非 Montgomery 规约, 因为其在非 SIMD 加速条件下加速比没有 Barrett 规约高, 适用性也普遍不如。

Barrett 模乘是一种近似算法。

取模显然有下列式子:

$$x \ mod \ q = x - \lfloor xs \rfloor \, q$$

其中 $s = \frac{1}{q}$，如果能以高精度求出 s，则此公式成立。

Barrett 模乘取 $r = \left\lfloor \frac{4^k}{q} \right\rfloor$，则使用近似 $\frac{r}{4^k} \approx \frac{1}{q}$，可得

$$x \ mod \ q = x - \left\lfloor \frac{xr}{2^k} \right\rfloor q$$

由于近似过程中 $\frac{r}{2^k}$ 与 $\frac{1}{q}$ 始终存在误差，由 $r = \left\lfloor \frac{2^k}{q} \right\rfloor$ 可得:

$$r = \frac{2^k}{q} - e \Rightarrow \frac{m}{2^k} = \frac{1}{q} - \frac{e}{2^k} \text{ for some } e \in [0, 1)$$

推出 $\frac{r}{2^k} \leq \frac{1}{q}$

默认 k 较大，一定满足 $x \leq 2^k$，则对于 $\lfloor \frac{x \cdot r}{2^k} \rfloor$，一定满足:

$$\lfloor \frac{x \cdot r}{2^k} \rfloor = \lfloor \frac{x}{q} - \frac{xe}{2^k} \rfloor \in \{ \lfloor \frac{x}{q} \rfloor, \lfloor \frac{x}{q} \rfloor - 1 \}$$

因此当前最终输出为

$$x \ mod \ q = x - \lfloor \frac{xr}{2^k} \rfloor q \in [x \bmod q, (x \bmod q) + q]$$

当 $x = a \times b$ 时，输出又可化为:

$$ab \ mod \ q = ab - a \lfloor \frac{br}{4^k} \rfloor q \in [b \bmod q, (b \bmod q) + q]$$

在应用 Barrett 模乘时一般取 $k = 32, 2^k = 2^{64}$，以提高近似精度和模数 q 的最大表示，由于 unsigned long long 的最大值为 $2^{64} - 1$，所以需要转化为 ___uint128，转化的过程会占据大量的时间开销，同时受此影响，如果模数是大模数，需要用到高精度类或者其他高精度处理方法 (因为只有这里会超过 ull)。

5.2 常规优化

很容易发现 mpi 优化与多线程的 pthread 优化十分相近，因此只需要将你在 pthread 里实现的 dif/dit 和 crt 合并在多进程里再实现一次即可，当然可以尝试将 crt 合并的模数分组计算，同一进程内使用多个线程进行计算，最后多个进程再统一合并。

当然需要注意如果实现了四分 NTT，需要对于奇数进行额外一次的变换才能继续四分。

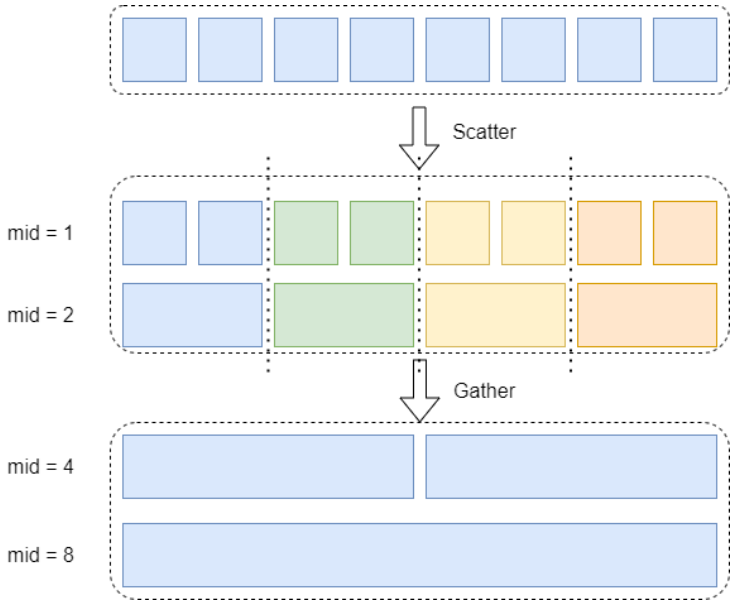

图 5.2: 多进程 NTT 的一种简要实现

6 口令猜测选题

MPI 编程在形式上和 pthread 没有太大区别，本次实验的重点是了解 MPI 编程的过程和细节，并且在工程上尝试对口令猜测算法进行并行化。

6.1 基础要求

按照先前多线程的基础要求，实现口令猜测算法的并行化。

6.2 进阶要求

1. 尝试使用多进程编程，在 PT 层面实现并行计算。先前的并行算法是对于单个 PT 而言，使用多进程/多线程进行并行的口令生成，**现在请尝试一次性从优先队列中取出多个 PT，并同时进行口令生成**。不需要实现加速，只需要在工程上加以实现即可。

   提示：需要注意每个 PT 生成之后均需要将产生的新 PT 放回优先队列，如果一次性取出多个 PT，那么等待各 PT 生成完成后，再将一系列新的 PT 挨个放回优先队列。

2. 尝试使用多进程编程，在进行口令猜测的同时，利用新的进程进行口令哈希。

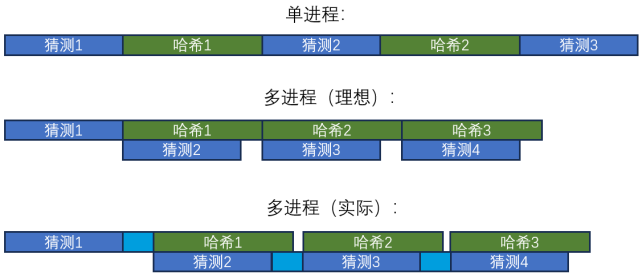

图 6.3: 利用多进程编程，在进行口令猜测的同时，利用新进程对口令进行哈希。实际情况中，进程间的通讯和 workload 传递会产生一定的开销。

如图6.3所示，先前的口令猜测/哈希过程是串行的，也就是猜测完一批口令之后，对这些口令进行哈希，哈希结束之后再继续进行猜测，周而复始。如果采用多进程（多线程理论上也可以）编程，就可以在猜

测完一批口令之后，对这批口令进行哈希，但同时继续进行新口令的生成。第一轮口令哈希结束、第二轮猜测结束之后，再同时进行第二轮口令哈希、第三轮口令猜测。

上述过程是否与计算机组成原理里面的流水线有点相似？可以尝试实现一下。不需要实现加速，只需要在工程上实现即可。

注意：临近期末，不要在这个进阶要求上死磕，见好就收。即便没有完全实现，也可以将实现的过程和遇到的困难记录在实验报告里。

上述进阶要求供大家参考。如有其它额外工作，将根据难度、工作量进行给分。