南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计

高斯消去 MPI 编程实验

田翔宇 2011748

年级：2020 级

专业：计算机科学与技术

2023 年 6 月 4 日

摘要

在 ARM 平台开展普通高斯消去计算的基础 MPI 并行化实验。探讨与多线程和 SIMD 的结合。

实验代码已上传 GitHub。

关键字：高斯消去 MPI 并行程序设计

目录

一、 问题重述 **1**

二、 实验环境 **1**

三、 实验设计 **1**

四、 实验结果分析 **2**

（一） x86 2

（二） arm 4

一、 问题重述

- 高斯消去法是一种求解线性方程组的算法。其本质是将线性方程组的增广矩阵转化为行阶梯矩阵。
- 高斯消去的基本思想是：先逐级消去变量，将原方程变换为同解的上三角方程组，此过程称为消元过程。然后按方程组的相反顺序求解上三角方程组便得到原方程组的解，此过程称为回代过程。
- LU 分解是矩阵分解的一种，将一个矩阵分解为一个下三角矩阵和一个上三角矩阵的乘积，有时需要再乘上一个置换矩阵。
- LU 分解可以被视为高斯消元法的矩阵形式。在数值计算上，LU 分解经常被用来解线性方程组、且在求逆矩阵和计算行列式中都是一个关键的步骤。
- 计算过程主要包括：
  1. 对于当前的消元行，应全部除以行首元素，使得本行的首元素为 1
  2. 对于之后的每一行，用本行减去当前的消元行，得到本次的消元结果

二、 实验环境

环境如下：

- 平台：x86
- CPU 型号：Intel(R) Core(TM) i7-10710U
- CPU 频率：1.10GHz
- L1 cache：384KB
- L2 cache：1.5MB
- L3 cache：12.0MB

三、 实验设计

设计思路：

- 对于高斯消去的两个阶段（行内除法与行间减法），每个阶段内工作基本一致，除了起始位置不一样，十分契合并行思想。
- 对于第一阶段行内除法：
  1. 将首元素与其他元素的除法分配给不同的进程，由于数据之间无依赖关系，无需过多通讯开销
  2. 将任务分配给各个进程后，进程内部可分配多条线程，由于数据之间无依赖关系，无需过多通讯开销

3. 实际体验下来，由于这个问题规模相对较小，线程状态切换时间开销相较于要处理的问题而言占比较高，不适合进行多线程并行处理，但是仍可以结合 SIMD 的向量化处理

- 对于第二阶段行间减法：

1. 将消去的行分给不同的进程，由于数据之间无依赖关系，无需过多通讯开销
2. 将计算任务分配给各个进程后，进程内部可分配多条线程，由于数据之间无依赖关系，无需过多通讯开销

数据划分：

- MPI 为进程级并行，负载不均影响相比于线程更大，应合理选择数据划分方式。
- 块划分：给各进程分配连续的几行数据。由于高斯消去的特点，优先完成除法的行不会参与后续计算，会导致负载不均，易造成负责前面几行的进程处于闲置状态，而负责后面的进程压力较大
- 循环划分：按照进程数量等步长分配数据。相比于块划分，不会导致严重的负载不均现象。

并行处理：

- 每次迭代内部的除法与消元减法有着严格的先后顺序，即除法操作之后才可消元。所以除法减法交替中间均需要进行同步。
- 使用单个线程分发任务，每次除法前检查任务状态，确认为当前任务，除法后广播结果实现通信。
- MPI 为进程级并行，进程内可以使用线程级和指令集的并行优化，结合 OpenMP 以及 SIMD。

问题规模和进程数量：

- 进程间通信的开销大于简单计算的开销，故当问题规模较小时进程间通信的开销会极大抵消掉并行优化，甚至会更慢。
- 问题规模增大到一定程度时，进程通讯开销占比下降到一定数量，并行优化才会显现出结果。

四、 实验结果分析

(一) x86

测量在不同任务规模下，串行算法，MPI 优化，MPI+SIMD 优化，MPI+OpenMP 优化以及 MPI+SIMD+OpenMP 优化的时间性能表现, 如表1。

为了能够更加直观的观察算法的性能表现随问题规模的变化情况，利用测量的数据计算了四种并行优化算法的加速比随时间的变化情况，如表2所示, 如图1。

表 1: x86 时间性能

N	serial	MPI_block	MPI_SIMD	MPI_OMP	MPI_OMP_SIMD
100.00	1.64	2.40	5.16	6.38	6.70
200.00	12.52	6.14	10.11	13.58	12.92
300.00	41.84	15.16	17.01	23.60	21.53
400.00	98.94	29.82	26.66	43.15	40.31
500.00	193.40	53.36	39.96	59.04	51.90
600.00	336.78	85.72	56.66	80.44	65.97
700.00	534.30	127.10	78.56	103.44	85.54
800.00	792.94	193.51	105.94	134.60	109.95
900.00	1130.92	265.65	142.08	183.35	131.90
1000.00	1549.75	365.86	181.19	217.02	162.68
1500.00	5227.93	1248.94	587.47	587.94	400.51
2000.00	12547.30	2855.66	1231.45	1220.32	775.82
2500.00	24559.00	5482.94	2339.07	2357.57	1335.25
3000.00	42296.90	9959.27	3978.63	3689.22	2179.09

表 2: x86 加速比

N	MPI	MPI_SIMD	MPI_OMP	MPI_OMP_SIMD
100	0.654913801	0.303369697	0.245022717	0.233495717
200	2.028857599	1.23213544	0.917239653	0.964596769
300	2.759097324	2.459982596	1.773114087	1.943359856
400	3.318051028	3.710576649	2.292986412	2.45455831
500	3.624585583	4.839549979	3.275523632	3.726466446
600	3.928664334	5.943491458	4.186727067	5.105170929
700	4.20389312	6.801414007	5.165094157	6.24594357
800	4.097627525	7.484694311	5.891142513	7.211787176
900	4.25714845	7.959740991	6.168060169	8.5743963
1000	4.235956223	8.55336505	7.141079813	9.526663593
1500	4.185893638	8.899043527	8.891975004	13.05331256
2000	4.393835401	10.18904543	10.28197522	16.17301503
2500	4.479166287	10.49947201	10.417082	18.39281034
3000	4.246987982	10.63102123	11.46499802	19.42166938

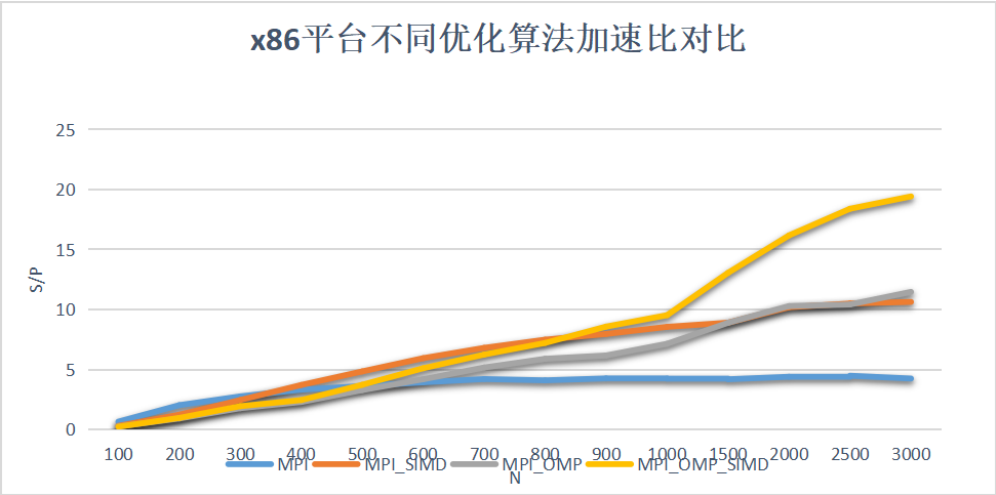

图 1: x86 加速比

数据分析:

- **随着问题规模的增加，四种并行优化算法的加速比都呈现一个递增的趋势**
- **MPI 单独优化，开了 8 条线程，但是实际加速比只是接近 4.5，主要是由于通信开销以及其余没有进行多进程并行的部分**
- **SIMD 采用了 4 路向量化的手段，理论上应该是 MPI 单独优化的 4 倍，实际上只是 2 到 2.5 倍，主要原因是其余没有进行 SIMD 优化的部分**
- **OpenMP 开了 8 条线程，但是实际加速比只是 MPI 的不到 3 倍，因为 MPI 本身的并行将任务大致划分为了 8 份，在 3000 规模的情况下，单个 OpenMP 的应对规模也不到 400，在该情况下，OpenMP 的优化很容易被抵消**
- **MPI 和 SIMD、OpenMP 融合到一起，在问题规模较大时性能提升接近 20 倍，与三者单独优化提升的乘积基本一致**

(二) arm

测量在不同任务规模下，串行算法，MPI 优化，MPI+SIMD 优化，MPI+OpenMP 优化以及 MPI+SIMD+OpenMP 优化的时间性能表现, 如表3。

为了能够更加直观的观察算法的性能表现随问题规模的变化情况, 利用测量的数据计算了四种并行优化算法的加速比随时间的变化情况，如表4所示。

表 3: arm 时间性能

N	serial	MPI_block	MPI_cycle	MPI_pipeline	MPI_SIMD	MPI_OMP	MPI_OMP_SIMD
100	0.33862	0.90476	3.77945	3.65523	3.5666	3.91236	4.06474
200	2.7185	3.72827	10.1691	10.3002	10.0733	10.5972	10.7221
300	9.10204	10.4317	20.3652	20.3559	19.5031	20.9339	19.8682
400	21.4773	21.3106	42.0001	41.9716	39.2503	41.1116	38.5392

Continued on next page

表 3: arm 时间性能 (Continued)

500	42.0603	38.6061	65.8516	65.8022	60.0661	62.6272	57.0654
600	72.9207	63.2224	96.6892	96.792	83.653	86.4704	75.4285
700	116.331	97.8989	134.915	135.279	112.542	111.4	92.7622
800	174.761	143.845	185.646	185.523	147.853	136.999	114.965
900	250.14	203.481	248.521	247.211	191.613	164.244	139.9
1000	346.754	276.217	316.976	315.751	245.347	212.05	166.701
1500	1176.88	912.376	891.533	893.707	679.538	456.254	339.622
2000	2833.69	2107.88	1941.73	1944.59	1435.86	896.433	603.925
2500	5568.77	4081.31	3571.36	3571.11	2603.29	2083.57	1176.75
3000	9726.77	7002.86	5951.89	5948.32	4320.35	2268.49	1387.72

表 4: arm 加速比

N	MPI_block	MPI_cycle	MPI_pipeline	MPI_SIMD	MPI_OMP	MPI_OMP_SIMD
100	0.374264998	0.089595047	0.092639861	0.094941962	0.08655134	0.083306681
200	0.729158564	0.267329459	0.263926914	0.269871839	0.256530027	0.25354175
300	0.872536595	0.44694086	0.447145054	0.466697089	0.434799058	0.458121018
400	1.007822398	0.511363068	0.511710299	0.547188174	0.522414598	0.557284531
500	1.089472907	0.63871341	0.639192915	0.700233576	0.671597964	0.737054327
600	1.153399744	0.754176268	0.753375279	0.871704541	0.843302448	0.96675262
700	1.188276886	0.862254012	0.859933914	1.033667431	1.044263914	1.254077631
800	1.214925788	0.941366903	0.94199102	1.181991573	1.275637048	1.520123516
900	1.229303964	1.00651454	1.011848178	1.305443785	1.522978008	1.787991422
1000	1.255368062	1.093944021	1.098188129	1.413320725	1.635246404	2.0800955
1500	1.289906793	1.320063307	1.316852167	1.731882544	2.579440399	3.465264323
2000	1.344331746	1.459363557	1.457217203	1.973514131	3.16107283	4.692122366
2500	1.36445651	1.559285538	1.559394698	2.139127796	2.672705981	4.732330571
3000	1.388971078	1.634232151	1.635212968	2.251384726	4.287772924	7.00917332

数据分析：

- **数据划分对比:**
  - **采用块划分的方式，随着消元的推进，负责前面行的进程会逐渐空闲下来，因此在后续的计算过程中，实际工作的进程会越来越少**
  - **循环划分则，达到了比较好的负载均衡，充分利用了每个线程的计算资源，因此其加速比逐步更加逼近理论加速比 2**

- **通信对比**:
  - **随着问题规模的增加，采用** pipeline **方式可以提高性能瓶颈，普通的通信方式加速比只能够达到** 1.2 **左右，而采用了** pipeline **的加速比可以达到** 1.6 **左右**
- **随着问题规模的增加，四种并行优化算法的加速比都呈现一个递增的趋势**
- MPI **单独优化，采用了** 2 **个进程进行多进程并行，最后根据不同的划分方式，稳定在** 1.3 **到** 1.6 **之间**
- SIMD **采用了** 4 **路向量化的手段，理论上应该是** MPI **单独优化的** 4 **倍，实际上只是** 1.5 **倍，主要原因是其余没有进行** SIMD **优化的部分**
- OpenMP **开了** 8 **条线程，但是实际加速比只是** MPI **的不到** 3 **倍，扩大而不断变大**
- MPI **和** SIMD、OpenMP **融合到一起，在问题规模较大时性能提升相接近** 7 **倍，与三者单独优化提升的乘积基本一致**