
南開大學
Nankai University

计算机学院

并行程序设计实验报告

# 基于 SIMD 的高斯消元并行实验

姓名：林逸典

学号：2213917

专业：计算机科学与技术

2024 年 4 月 18 日

# 目录

**1 实验背景** **2**
1.1 算法概述 . . . 2
1.2 SIMD 编程技术 . . . 2

**2 实验环境** **2**
2.1 实验平台 . . . 2
2.2 实验数据集 . . . 2

**3 普通高斯消元** **3**
3.1 Neon 算法 . . . 3
3.2 实验结果 . . . 4
3.2.1 并行算法的性能分析 . . . 4
3.2.2 内存对齐的性能分析 . . . 5
3.2.3 不同部分的性能分析 . . . 6
3.2.4 总结 . . . 7

**4 状态压缩 Buchberger 算法的性能研究** **8**
4.1 Neon 算法 . . . 8
4.2 并行算法的性能分析 . . . 8
4.3 Profiling 分析 . . . 9
4.4 汇编语言分析 . . . 10
4.5 总结 . . . 10

**5 总结** **11**

# 1 实验背景

本实验隶属于算法导论与并行程序设计系列研究，旨在通过 SIMD 编程技术并行化普通高斯消元法和状态压缩 Buchberger 算法，并探究其优化效果。本实验代码已上传GitHub。

## 1.1 算法概述

普通高斯消元法，作为求解线性方程组的经典算法，通过一系列初等行变换将增广矩阵转化为上三角形式，进而通过回带求解得出最终结果。

Grobner 基是多项式环中的一组特殊生成元，Buchberger 算法作为计算 Grobner 基的核心方法，其本质可视为欧几里得算法和线性系统中高斯消元法的泛化。我们之前提出的状态压缩 Buchberger 算法在时间和空间上均进行了优化。

有关这两种算法的深入分析与讨论，可查阅我们在GitHub上发布的相关论文及代码。

## 1.2 SIMD 编程技术

SIMD（单指令多数据）技术，作为一种高效的并行计算手段，通过向量寄存器同时处理多个数据元素，利用单条指令实现数据的并行操作。这种技术能够显著提升数据处理的效率，减少指令周期数，从而增强计算机系统的性能。

SIMD 编程的核心优势在于其强大的数据并行处理能力。通过一次性处理多个数据元素，SIMD 能够在单个指令周期内完成更多的计算任务，显著提高计算密集型应用的性能。在算法优化领域，SIMD 技术已成为提升算法执行效率的重要手段之一。

# 2 实验环境

## 2.1 实验平台

本实验采用华为鲲鹏 920 处理器作为核心计算平台，其基于 ARM 架构，CPU 主频高达 2.6GHz。该平台提供了稳定可靠的计算环境，为实验的顺利进行奠定了坚实基础。此外，实验平台配备了 191GB 的内存（RAM），确保了在处理大规模数据集时的高效性能。

| | |
|---|---|
| 处理器架构 | ARM |
| CPU 型号 | 华为鲲鹏 920 处理器 |
| CPU 主频 | 2.6GHz |
| 内存（RAM） | 191GB |

表 1: 实验环境配置

## 2.2 实验数据集

为了全面评估普通高斯消元法与状态压缩 Buchberger 算法的性能，我们准备了两组不同的数据集。

普通高斯消元法的数据集是通过程序随机生成的，矩阵列数 n 为 $2^m$,m 称为问题规模，确保了数据的唯一性和解的存在性。这些数据集涵盖了不同规模的问题，以充分检验算法在不同情况下的表现。

| test | 问题规模 m | 矩阵行数 n |
|---|---|---|
| test1 | 7 | 128 |
| test2 | 8 | 256 |
| test3 | 9 | 512 |
| test4 | 10 | 1024 |
| test5 | 11 | 2048 |

表 2: 内存不对齐与对齐的性能对比

状态压缩 Buchberger 算法所采用的数据集则来源于相关研究，并经过严格的格式规范化和数据清洗处理。数据集的准确性与可靠性得到了充分保障。该数据集包含多个精心设计的测试样例，每个样例具有不同的消元矩阵列数、消元子行数和消元行行数，旨在全面评估算法在不同规模和复杂度问题上的性能。

以下是状态压缩 Buchberger 算法实验数据集的详细统计信息：

| **测试编号** | test1 | test2 | test3 | test4 | test5 | test6 | test7 | test8 | test9 | test10 | test11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 消元矩阵列数 | 130 | 254 | 562 | 1011 | 2362 | 3799 | 8399 | 23045 | 37960 | 43577 | 85401 |
| 消元子行数 | 22 | 106 | 170 | 539 | 1226 | 2759 | 6375 | 18748 | 29304 | 39477 | 5724 |
| 消元行行数 | 8 | 53 | 53 | 263 | 453 | 1953 | 4345 | 14325 | 14921 | 54274 | 756 |

表 3: 状态压缩 Buchberger 算法实验数据集

通过这组精心构造的数据集，我们能够全面评估状态压缩 Buchberger 算法在不同问题规模和复杂度下的性能表现，为算法的进一步优化和应用提供有力支持。

# 3 普通高斯消元

在本部分，我们针对普通高斯消元法进行了详细的性能研究，主要聚焦于串行算法与并行算法的时间对比，包括内存对齐与不对齐的性能差异，以及前向消元和回带求解的串行与并行实现的时间对比。所有测试均以微秒为单位记录时间，并计算了不同情况下的加速比。

## 3.1 Neon 算法

为了提升高斯消元法的性能，我们采用了 NEON 指令集对前向消元和回带求解进行了并行化优化。以下是优化后的核心代码段。

前向消元并行化核心部分

```
// 行加减运算（NEON版本）
for(int i=I+1;i<=N;i++)
{
    K=vdupq_n_f32(C[i][I]);
    C[i][I]=0;
    for(int j=I+1;j<=N+5;j+=4)
    {
        C_I=vld1q_f32(&C[I][j]);
        C_i=vld1q_f32(&C[i][j]);
        kC_I=vmulq_f32(K,C_I);
        C_i=vsubq_f32(C_i,kC_I);
```

```
            vst1q_f32(&C[i][j],C_i);
        }
    }
```

回带求解并行化核心部分

```
for (int i=I-4;i>=1;i=i-4)
{
    for(int j=0;j<=3;j++)
    {
     C_j[j]=C[i+j][I];
     C_n1[j]=C[i+j][N+1];
    }
    K=vdupq_n_f32(C[I][N+1]);//复制第I行解(系数K)
    C_N1=vld1q_f32(&C_n1[0]);//复制第N+1列的值
    C_I=vld1q_f32(&C_j[0]);//复制第I列的值
    kC_I=vmulq_f32(K,C_I);//将第N+1列与第I列相乘
    C_N1=vsubq_f32(C_N1,kC_I);//将第N+1列减去(第N+1列与第I列相乘)
    vst1q_f32(&C_n1[0],C_N1);
    for(int j=0;j<=3;j++)
      {
          C[i+j][I]=0;
          C[i+j][N+1]=C_n1[j];
      }
}
//边界处理
y=1;
while(C[y][I]!=0&&y<I)
{
  C[y][N+1]=C[y][N+1]-C[I][N+1]*C[y][I];
  C[y][I]=0;
  y++;
}
```

在上述优化后的代码中，我们充分利用了 NEON 指令集的 SIMD 特性，对向量进行了加载、运算和存储操作，从而减少了循环次数和内存访问次数，提高了算法的执行效率。

## 3.2 实验结果

### 3.2.1 并行算法的性能分析

在本节中，我们对串行算法与并行算法的运行时间进行了详细的测试和对比分析，并计算了加速比。实验结果以表格形式呈现如下：

| 问题规模 m | 矩阵行数 n | 串行算法 | 并行算法 | 加速比 |
|---|---|---|---|---|
| 7 | 128 | 4788.71 | 3864.75 | 1.24 |
| 8 | 256 | 36776.6 | 28557.6 | 1.29 |
| 9 | 512 | 292481 | 220245 | 1.33 |
| 10 | 1024 | 2357154 | 1756869 | 1.34 |
| 11 | 2048 | 18841196 | 14086434 | 1.34 |

表 4: 普通高斯消元：串行算法与并行算法性能对比

为了更直观地展示并行算法与串行算法的加速比，我们还绘制了加速比随问题规模变化的折线图，如图3.1所示。

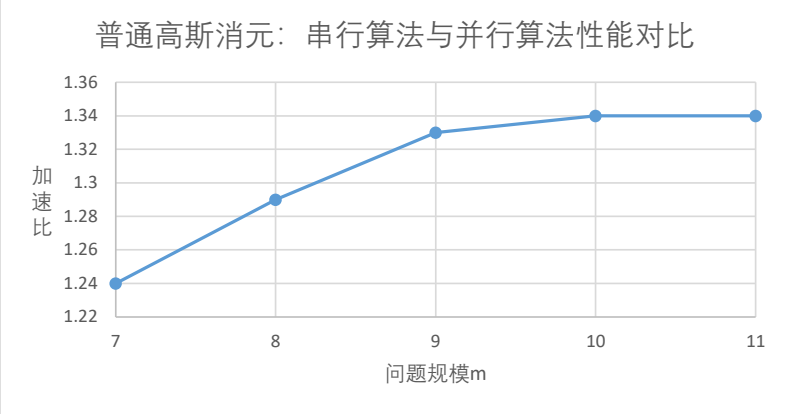


图 3.1: 普通高斯消元：串行算法与并行算法性能对比

从实验结果可以看出，并行算法在运行效率上明显优于串行算法。随着问题规模的增大，加速比逐渐提高，表明并行算法在处理大规模问题时更具优势。这一结果验证了并行化策略的有效性，并为后续的研究提供了有价值的参考。

### 3.2.2 内存对齐的性能分析

在本节中，我们深入探讨了内存对齐对并行算法性能的影响。通过对比内存对齐与不对齐情况下算法的运行时间，我们分析了其对加速比的作用。以下是我们收集的实验数据：

| 问题规模 m | 矩阵行数 n | 内存不对齐 | 内存对齐 | 加速比 |
|---|---|---|---|---|
| 7 | 128 | 3864.75 | 3853.42 | 1.00 |
| 8 | 256 | 28557.6 | 28533.8 | 1.00 |
| 9 | 512 | 220245 | 222247 | 0.99 |
| 10 | 1024 | 1756869 | 1795635 | 0.98 |
| 11 | 2048 | 14086434 | 14390553 | 0.98 |

表 5: 普通高斯消元：内存不对齐与对齐的性能对比

从表中数据可以看出，在大多数情况下，内存对齐并没有带来显著的性能提升。在较小规模的问题中，内存对齐与不对齐的性能差距微乎其微；随着问题规模的增大，尽管内存对齐的耗时略有增加，但整体加速比仍然接近 1，表明对齐操作并未带来明显的性能优势。

这一结果可能归因于现代计算机硬件对内存访问的优化机制。即使不进行显式的内存对齐，硬件通常能够高效地处理非对齐的内存访问。因此，在实际应用中，内存对齐可能并不是提高算法性能的关键因素。

综上所述，我们的实验结果表明，在本研究的背景下，内存对齐对并行算法的性能影响有限。

### 3.2.3 不同部分的性能分析

在本节中，我们详细分析了算法中不同部分的并行性能，特别是前向消元与回带求解两个关键步骤。通过对比它们的串行与并行实现，我们深入探讨了各自在不同问题规模下的加速效果。

表6展示了前向消元部分在串行和并行实现下的性能对比。从表中数据可以看出，随着问题规模的增大，并行算法相对于串行算法的运行时间显著减少，加速比逐渐提高。这表明前向消元部分在并行化后取得了良好的性能提升。

| 问题规模 m | 矩阵行数 n | 串行算法 | 并行算法 | 加速比 |
|---|---|---|---|---|
| 7 | 128 | 4699.02 | 3704.65 | 1.27 |
| 8 | 256 | 36413.4 | 27902.7 | 1.31 |
| 9 | 512 | 290964 | 217528 | 1.34 |
| 10 | 1024 | 2350156 | 1745313 | 1.35 |
| 11 | 2048 | 18809054 | 14036462 | 1.34 |

表 6: 普通高斯消元：前向消元串行算法与并行算法的性能对比

然而，表7中回带求解部分的性能数据却产生了不同的表现，并行算法起到了负优化的效果。在串行与并行实现下，随着问题规模的增大，回带求解部分的运行时间均有所增加，但并行算法的加速比却出现了负优化现象。这可能是由于回带求解部分本身的计算复杂度较低（通常为 $O(n^2)$）导致并行化带来的性能提升有限，甚至可能由于并行化引入的额外开销而降低效率。

| 问题规模 m | 矩阵行数 n | 串行算法 | 并行算法 | 加速比 |
|---|---|---|---|---|
| 7 | 128 | 89.6900 | 160.100 | 0.55 |
| 8 | 256 | 363.140 | 654.900 | 0.55 |
| 9 | 512 | 1517.12 | 2717.58 | 0.56 |
| 10 | 1024 | 6998.59 | 11556.3 | 0.61 |
| 11 | 2048 | 32141.7 | 49971.4 | 0.64 |

表 7: 普通高斯消元：回带求解串行算法与并行算法的性能对比

图3.2直观地展示了前向消元与回带求解部分在不同问题规模下的加速比对比。可以看出，随着问题规模的增大，前向消元部分的加速比逐渐上升，而回带求解部分的加速比也逐渐上升但表现为负优化。这进一步验证了前向消元部分在并行化后具有较好的性能提升，而回带求解部分则相对较难通过并行化获得显著的性能提升。

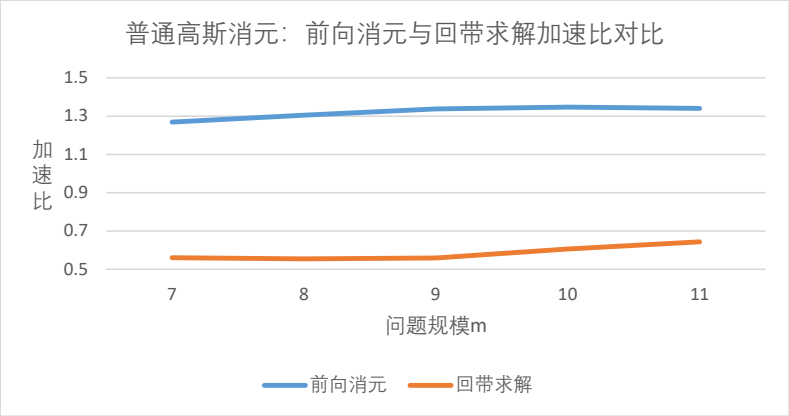


图 3.2: 不同部分加速比对比

综上所述，针对算法中不同部分的并行性能分析表明，前向消元部分在并行化后能够获得较好的性能提升，而回带求解部分则相对较难通过并行化获得显著的性能提升。

### 3.2.4 总结

在本部分中，我们针对普通高斯消元法进行了深入的性能研究，主要聚焦于串行算法与并行算法的时间对比，以及内存对齐与不对齐对性能的影响。此外，我们还特别关注了前向消元和回带求解这两个关键步骤在串行与并行实现下的性能差异。

首先，通过对比串行算法与并行算法的运行时间，我们发现并行算法在运行效率上表现出明显的优势。随着问题规模的增大，并行算法相对于串行算法的运行时间显著减少，加速比逐渐提高。这一结果充分验证了并行化策略在普通高斯消元法中的有效性，为处理大规模线性方程组提供了高效的计算手段。

然而，在内存对齐与不对齐的性能对比中，我们并未观察到显著的性能差异。即使在较大规模的问题中，内存对齐与不对齐的耗时差距也微乎其微，整体加速比接近 1。这一结果可能归因于现代计算机硬件对内存访问的优化机制，使得非对齐内存访问也能得到高效处理。因此，在实际应用中，我们无需过分关注内存对齐问题，而应将更多的精力投入到算法本身的优化上。

接下来，我们重点关注了前向消元与回带求解两个步骤的性能表现。实验结果表明，前向消元部分在并行化后取得了显著的性能提升。随着问题规模的增大，并行算法的运行时间显著减少，加速比逐渐提高。这一结果符合预期，因为前向消元过程中存在大量的行变换操作，这些操作在并行环境下可以更有效地进行。

然而，回带求解部分的性能表现却不尽如人意。在串行与并行实现下，随着问题规模的增大，回带求解部分的运行时间均有所增加，且并行算法出现了负优化现象。这可能是由于回带求解步骤本身的计算复杂度相对较低，导致并行化带来的性能提升有限。此外，并行化可能引入了额外的通信和同步开销，进一步降低了并行算法的效率。

综上所述，我们的研究表明普通高斯消元法中的前向消元部分在并行化后具有显著的性能提升潜力，而回带求解部分则相对较难通过并行化获得显著的性能提升。在未来的研究中，我们将进一步探索其他并行化方法对回带求解部分的优化策略，以期提高整体算法的性能。

# 4 状态压缩 Buchberger 算法的性能研究

在本节中，我们对状态压缩 Buchberger 算法进行了深入的性能研究。研究的核心内容主要包括串行算法与并行算法的执行时间对比，以及不同算法的内存消耗分析。实验数据以秒为单位记录，同时计算了不同情况下的加速比，以全面评估算法的性能。

## 4.1 Neon 算法

为了提升高斯消元法的性能，我们采用了 NEON 指令集对前向消元进行了并行化优化。以下是优化后的核心代码段。

前向消元并行化核心部分

```
for(int i=0;i<col2;i++)
{
  if(D[i]==0) continue;
  if((B[i][nowrow]&nownum)==0) continue;
  for(int j=nowrow;j<comrow+8;j+=8)
  {
    A_8=vld1q_u16(&A[I][j]);
    B_8=vld1q_u16(&B[i][j]);
    B_8=veorq_u16(A_8,B_8);
    vst1q_u16(&B[i][j],B_8);
  }
}
```

上述代码中，我们利用了 NEON 指令集的并行处理能力，对前向消元过程中的行变换操作进行了优化。通过加载 8 个连续的 unsigned short 整数到 NEON 寄存器中，并使用位异或操作实现行变换，然后将结果写回到内存中，实现了对多个元素的并行处理。

## 4.2 并行算法的性能分析

表8详细展示了前向消元部分在串行和并行实现下的时间性能对比。从表中数据可以清晰地观察到，随着问题规模的逐渐增大，并行算法相较于串行算法在运行时间上呈现出显著的减少趋势，同时加速比也在稳步提高。这一结果充分验证了并行化策略在前向消元部分中的有效性，显著提升了算法的计算效率。

| test | 串行算法 | 并行算法 | 加速比 |
|---|---|---|---|
| test1 | 0.0082 | 0.0076 | 1.07 |
| test2 | 0.2181 | 0.1708 | 1.28 |
| test3 | 0.4805 | 0.3235 | 1.49 |
| test4 | 13.232 | 6.9099 | 1.91 |
| test5 | 106.63 | 47.017 | 2.27 |
| test6 | 1540.1 | 697.29 | 2.21 |
| test7 | 20489 | 9162.0 | 2.24 |
| test8 | 385664 | 160313 | 2.41 |

表 8: 状态压缩 Buchberger 算法：串行算法与并行算法的时间性能对比

图4.3则更为直观地展示了不同测试案例下并行算法相较于串行算法的加速比变化。随着问题规模的增加，加速比呈现出稳步上升的趋势，进一步验证了并行化策略的有效性和优越性。

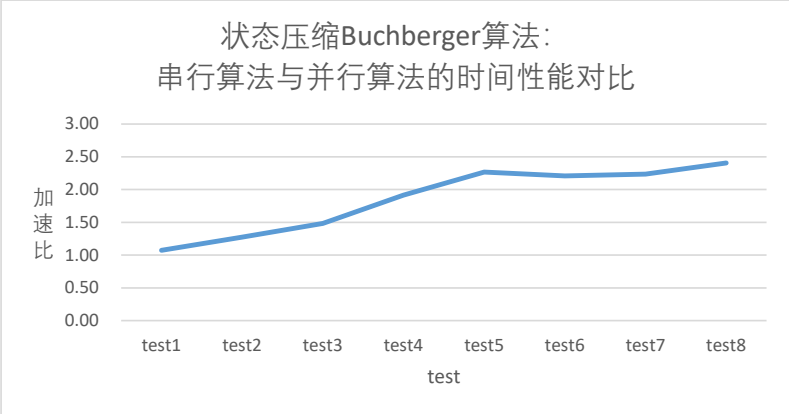


图 4.3: 状态压缩 Buchberger 算法：串行算法与并行算法的时间性能对比

表9详细展示了前向消元部分在串行和并行实现下的空间性能对比。从表中数据可以清晰地观察到，随着问题规模的增大，并行算法的空间占比与串行算法逐渐接近，甚至在某些情况下几乎相同。这一结果表明，并行算法在提升计算效率的同时，并未造成过度的内存消耗，从而确保了算法的空间效率。

| | 串行算法 | 并行算法 | 空间占比 |
|---|---|---|---|
| test7 | 18560 | 18944 | 0.98 |
| test8 | 110912 | 112640 | 0.98 |
| test9 | 251648 | 253312 | 0.99 |
| test10 | 527232 | 531648 | 0.99 |
| test11 | 904960 | 908992 | 1.00 |

表 9: 状态压缩 Buchberger 算法：串行算法与并行算法的空间性能对比

这一结果进一步验证了并行化策略在前向消元部分中的有效性和可行性。通过采用并行化策略，我们不仅能够提高算法的计算效率，还能保持较低的内存消耗。

## 4.3 Profiling 分析

为了具体探讨 SIMD 编程对状态压缩 Buchberger 算法运算速度的提升效果，并探讨其未能达到理论最大加速比 4 的原因，我们进行了详细的 profiling 分析。本部分将以 test7 为例，分别探讨 CPI（每条指令的时钟周期数）和各级缓存命中率的影响。

**CPI 分析** 表10展示了串行算法和并行算法的时钟周期数、指令数以及 CPI 的对比。从表中可以看出，串行算法的时钟周期数是并行算法的 2.12 倍，这表明 SIMD 编程通过减少时钟周期数有效提高了算法的运算效率。同时，串行算法的指令数是并行算法的 3.85 倍，这验证了单指令多数据方法能够显著减少指令数。然而，值得注意的是，尽管指令数有所减少，但并未达到理论上的最大减少比例 8，这可能是由于额外的加载开销等因素所致。此外，串行算法的 CPI 是并行算法的 0.55 倍，这表明 SIMD 指令相对于普通指令可能需要更长的执行时间。这也可能是导致加速比未能达到理论最大加速比 8 的一个重要原因。

| | 串行算法 | 并行算法 | 比值 |
|---|---|---|---|
| 时钟周期数 | 54617697997 | 25803069547 | 2.12 |
| 指令数 | 164484653041 | 42683459614 | 3.85 |
| CPI | 0.33 | 0.60 | 0.55 |

表 10: CPI 相关 profiling 分析

**缓存命中率分析** 表11展示了串行算法和并行算法的各级缓存命中率对比。从表中可以看出，串行算法与并行算法的各级缓存命中率相差不大。这表明 SIMD 编程对各级缓存的影响较小，并非通过提高缓存命中率来提高运行效率。因此，我们可以推断，SIMD 编程对算法性能的提升主要来源于其并行计算能力，而非对缓存利用率的优化。

| cache level | 串行算法 | 并行算法 |
|---|---|---|
| L1 cache | 99.88% | 99.60% |
| L2 cache | 53.15% | 54.55% |
| L3 cache | 98.66% | 99.44% |

表 11: 缓存相关 profiling 分析

## 4.4 汇编语言分析

为了深入剖析 SIMD 编程如何提升状态压缩 Buchberger 算法的运算速度，我们对其执行过程进行了汇编语言分析。本部分将详细探讨 SIMD 编程中的数据加载与存储以及并行计算部分的汇编代码。

**数据加载与存储** SIMD 编程通过特定的加载和存储指令，实现数据在内存与向量寄存器之间的快速传输。以下展示了数据加载与存储的汇编代码：

数据加载与存储汇编代码

```
vld1.16 {d16–d17}, [r3] ; 从内存地址r3加载16位数据到向量寄存器d16和d17
vst1.64 {d16–d17}, [r3:64] ; 将向量寄存器d16和d17中的64位数据存储到内存地址r3
```

这些指令能够一次性处理多个数据元素，从而显著提高数据吞吐量。然而，这种并行加载和存储机制也带来了额外的开销。

**并行计算** SIMD 编程的核心优势在于其强大的并行计算能力。以下是本算法并行计算部分的汇编代码，展示了如何在单个指令中对多个数据元素执行相同的运算：

并行计算汇编代码

```
veor q8, q9, q8 ; 对向量寄存器q8和q9中的元素执行异或运算，结果存储在q8中
```

通过利用 SIMD 指令集，我们在单个周期内对多个数据项执行相同的操作，从而有效提高计算效率。

## 4.5 总结

本部分围绕状态压缩 Buchberger 算法的性能进行了深入的研究，重点关注了算法在串行和并行实现下的时间与空间性能。经过一系列实验与数据分析，我们得出以下结论。

在时间性能方面，状态压缩 Buchberger 算法的 SIMD 并行实现在处理大规模问题时展现出了显著的优势。通过 profiling 分析，我们发现 SIMD 编程显著减少了算法的时钟周期数和指令数，从而提高了运算效率。特别是并行计算部分的汇编代码显示，SIMD 指令能够在单个周期内对多个数据元素执行相同的运算，这种并行计算能力有效提升了算法的执行速度。然而，我们也注意到，尽管指令数有所减少，但并未达到理论上的最大减少比例，这可能是由于额外的数据加载与存储开销等因素所致。此外，串行算法的 CPI 较低，表明 SIMD 指令相对于普通指令可能需要更长的执行时间，这也是加速比未能达到理论最大值的原因之一。尽管如此，SIMD 并行化策略在状态压缩 Buchberger 算法中仍然显著提升了计算效率，特别适用于处理大规模问题。

在空间性能方面，我们的研究表明，状态压缩 Buchberger 算法的 SIMD 并行实现并未导致过度的内存消耗。即使在处理大规模问题时，并行算法的空间占比与串行算法也相当接近。这得益于 SIMD 编程通过向量寄存器高效地处理数据，而无需大量增加额外的内存开销。这一优点使得状态压缩 Buchberger 算法在内存资源有限的情况下仍能保持良好的性能。

在汇编语言分析方面，我们深入探讨了 SIMD 编程中的数据加载与存储以及并行计算部分的汇编代码。数据加载与存储指令实现了数据在内存与向量寄存器之间的快速传输，显著提高了数据吞吐量。然而，这种并行加载和存储机制也带来了一定的额外开销。在并行计算部分，我们观察到 SIMD 指令能够在单个指令中处理多个数据元素，从而实现了高效的并行计算。这些汇编层面的细节进一步证实了 SIMD 编程在提升状态压缩 Buchberger 算法性能方面的有效性。

综上所述，状态压缩 Buchberger 算法的 SIMD 并行实现在时间和空间性能上均表现出色。通过合理利用 SIMD 指令集的特性，我们能够有效提高算法的运算速度，为大规模问题的求解提供了有效的解决方案。然而，我们也认识到算法在加速比方面仍存在提升的空间。在未来的研究中，我们将进一步探索优化 SIMD 编程策略，以进一步提高算法的性能。

# 5 总结

本文围绕普通高斯消元法和状态压缩 Buchberger 算法的性能优化进行了系统研究，利用 SIMD 编程技术实现了算法的并行化，并通过实验与数据分析深入探讨了其性能特点。现将主要结论总结如下：

在普通高斯消元法的研究中，我们观察到并行化策略显著提升了算法的运行效率。随着问题规模的扩大，并行算法相较于串行算法的运行时间大幅减少，加速比逐渐提升，这充分验证了并行化策略的有效性。然而，我们也发现内存对齐与否对算法性能的影响并不显著，这主要归功于现代计算机硬件对内存访问的优化机制。此外，虽然前向消元部分在并行化后取得了显著的性能提升，但回带求解部分却并未展现出明显的性能改善，甚至出现了负优化现象。这表明在未来的研究中，我们需要针对回带求解部分探索更加高效的并行化方法和优化策略。

在状态压缩 Buchberger 算法的研究中，我们利用 SIMD 编程技术实现了算法的并行化，并通过 profiling 分析和汇编语言分析深入探讨了其性能特点。实验结果表明，SIMD 并行化显著提高了算法的运算速度，减少了时钟周期数和指令数。特别地，SIMD 指令的并行计算能力使得单个指令能够处理多个数据元素，从而有效提升了执行效率。尽管并未达到理论上的最大加速比，但 SIMD 并行化仍然为处理大规模问题提供了有效的解决方案。在空间性能方面，SIMD 并行实现并未导致过度的内存消耗，这使得 SIMD 并行优化更具有可行性。

综上所述，通过 SIMD 编程技术的应用，我们成功提升了普通高斯消元法和状态压缩 Buchberger 算法的性能。然而，我们也认识到在回带求解部分和加速比方面仍存在优化空间。在后续的实验中，我们将继续探索更加高效的并行化方法和优化策略，以进一步提高这两种算法的性能。