南開大學
Nankai University

计算机学院

并行程序设计实验报告

基于 Pthread 和 OpenMP 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 5 月 26 日

目录

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 Pthread 编程技术和 OpenMP 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 编程技术概述

Pthread 是 POSIX 线程（POSIX Threads）的简称，它是一组跨平台的 API，允许开发者在 C 和 C++ 等语言中编写多线程程序。pthread 提供了创建、同步、销毁和管理线程的功能，使得多线程编程更加容易和标准化。以下是 Pthread 的重要函数：

Pthread 的重要函数

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
    //创建一个新的线程。
int pthread_join(pthread_t thread, void **value_ptr);
    //等待指定的线程终止。
void pthread_exit(void *value_ptr);
    //终止调用它的线程。
int pthread_cancel(pthread_t thread);
    //请求取消指定的线程。
```

OpenMP 是一种支持共享内存并行编程的 API，它提供了一套编译指导语句和运行时库函数，使得程序员能够方便地在多核处理器或多线程环境中实现并行计算。通过 OpenMP，我们可以将串行代码中的循环、条件判断等部分进行并行化改造，从而充分利用多核处理器的计算资源，提高程序的执行效率。以下是 OpenMP 的重要函数：

OpenMP 的重要函数

```
#pragma omp parallel
    //创建一个并行区域，其中的代码将由多个线程并行
#pragma omp master
    //标记代码段只能由主线程执行。
#pragma omp critical
    //创建一个临界区，只允许一个线程同时执行。
#pragma omp parallel for
    //将for循环并行化，每个线程都会执行其中的一部分迭代。
```

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。实验环境配置如表1所示：

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。如表2所示，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: 普通高斯消元法实验数据集

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。

状态压缩 Buchberger 算法实验数据集的详细统计信息如表3所示：

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

在本部分中，我们将利用 Pthread 技术和 OpenMP 技术对普通高斯消元法进行并行化研究，并重点关注串行算法与并行算法在执行时间上的对比。所有测试均以微秒为单位记录时间，并计算加速比，以全面评估采用 Pthread 技术和 OpenMP 技术对算法性能的影响。

3.1 算法设计

3.1.1 Pthread 技术

我们采用 Pthread 技术对前向消元和回带求解过程进行了并行化优化。以下展示了优化后的核心代码段，这些代码使用多线程进行并行运算。

前向消元并行化核心部分

```
void* onethread_Forward_elimination(void *arg)
{
  Parameter para=*(Parameter*)arg;
  int I=para.I;
  int begin=para.I+para.Id;
  for(int i=begin;i<=N;i=i+MAX_THREADS)
  {
    pthread_mutex_lock(&mutex);
    //行加减运算
    k=C[i][I];
    C[i][I]=0;
    for(int j=I+1;j<=N+1;j++)
    {
       C[i][j]=C[i][j]-k*C[I][j];
    }
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(NULL);
}

void Forward_elimination()//前向消元
{
  for(int I=1;I<=N;I++)
  {
    //主元选择及交换
    pivot=abs(C[I][I]);
    pivotcol=I;
    for(int i=I;i<=N;i++)
    {
      if(pivotcol<abs(C[i][I]))
      {
        pivotcol=i;
        pivot=abs(C[i][I]);
      }
    }
```

```
    pivot=C[pivotcol][I];
    for(int j=I;j<=N+1;j++)
    {
      swap(C[pivotcol][j],C[I][j]);
      C[I][j]=C[I][j]/pivot;
    }
    //分配任务,创建线程
    for(int i=1;i<=MAX_THREADS;i++)
    {
      param[i].I=I;
      param[i].Id=i;
      pthread_create(&handles[i],NULL,
      onethread_Forward_elimination,&param[i]);
    }
    //挂起主线程
    for(int i=1;i<=MAX_THREADS;i++)
    {
      pthread_join(handles[i],NULL);
    }
  }
}
```

回带求解并行化核心部分

```
void* onethread_backband_solving(void *arg)
{
  //行加减运算
  Parameter para=*(Parameter*)arg;
  int I=para.I;
  int begin=para.I-para.Id;
  for(int i=begin;i>=1;i=i-MAX_THREADS)
  {
    pthread_mutex_lock(&mutex);
    C[i][N+1]=C[i][N+1]-C[I][N+1]*C[i][I];
    C[i][I]=0;
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(NULL);
}

void backband_solving()//回带求解
{
  for(int I=N;I>=1;I--)
  {
    //分配任务,创建线程
    for(int i=1;i<=MAX_THREADS;i++)
    {
      param[i].I=I;
      param[i].Id=i;
```

```
        pthread_create(&handles[i],NULL,
        onethread_backband_solving,&param[i]);
    }
    //挂起主线程
    for(int i=1;i<=MAX_THREADS;i++)
    {
        pthread_join(handles[i],NULL);
    }
  }
}
```

在上述代码中，通过 Pthreads 的一些关键函数创建、等待、退出和取消线程。我们实现了对算法的多线程并行化处理。这使得不同的行能够同时进行前向消元或回带求解，从而实现了多行的并行处理。

3.1.2 OpenMP 技术

我们采用 OpenMP 指令对前向消元和回带求解过程进行了并行化优化。以下展示了优化后的核心代码段，这些代码使用多线程进行并行运算。

前向消元并行化核心部分

```
//行加减运算
#pragma omp parallel for
for(int i=I+1;i<=N;i++)
{
  k=C[i][I];
  C[i][I]=0;
  for(int j=I+1;j<=N+1;j++)
  {
    C[i][j]=C[i][j]-k*C[I][j];
  }
}
```

回带求解并行化核心部分

```
#pragma omp parallel for
for(int I=N;I>=1;I--)
{
  for(int i=I-1;i>=1;i--)
     {
      C[i][N+1]=C[i][N+1]-C[I][N+1]*C[i][I];
      C[i][I]=0;
     }
}
```

在上述代码中，通过 #pragma omp parallel for 指令，我们实现了对循环的多线程并行化处理。这使得不同的行能够同时进行前向消元或回带求解，从而实现了多行的并行处理。

3.2 并行技术的性能分析

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尽的测试和对比分析，并计算了相应的加速比，其中对于并行算法，均在线程数为 4 的情况下进行测试。

3.2.1 Pthread 的性能分析

表4列出了在不同问题规模下，普通高斯消元法串行算法与 Pthread 并行算法的运行时间及加速比对比。

问题规模 m	矩阵行数 n	串行算法	Pthread 算法	加速比
7	128	4714.03	23504.0	0.20
8	256	36128.3	100010	0.36
9	512	288412	541402	0.53
10	1024	2436304	3351129	0.73
11	2048	18895642	22982793	0.82

表 4: 普通高斯消元：串行算法与 Pthread 算法性能对比

图3.1直观地展示了 Pthread 算法在不同问题规模下的加速比。从图中可以看出，尽管 Pthread 技术在普通高斯消元问题上未能展现出显著的加速效果（加速比始终低于 1），但随着问题规模的增大，加速比有所上升，表明在处理超大规模问题时，Pthread 算法或许能够减少执行时间，提升算法效率。

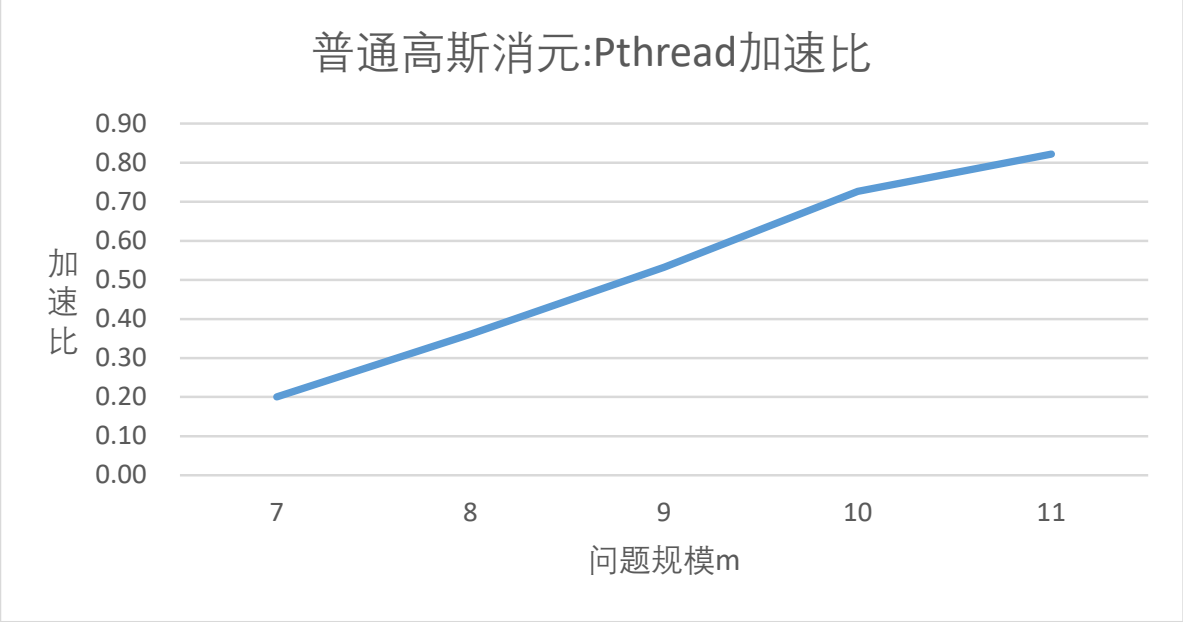

图 3.1: 普通高斯消元：串行算法与 Pthread 算法加速比

值得注意的是，由于 Pthread 技术的编程复杂度较高，我们在实现过程中采用了严格的线程锁操作以保证答案的正确性。这可能是导致 Pthread 技术未能充分发挥性能优势的重要原因。未来研究可以进一步探索更加复杂和精细的线程管理机制，以充分发挥 Pthread 技术的潜力。

3.2.2 OpenMp 的性能分析

表5列出了在不同问题规模下，普通高斯消元法串行算法与 OpenMp 并行算法的运行时间及加速比对比。

问题规模 m	矩阵行数 n	串行算法	OpenMp 算法	加速比
7	128	4714.03	101244	0.05
8	256	36128.3	71919.2	0.50
9	512	288412	303352	0.95
10	1024	2436304	1908469	1.28
11	2048	18895642	11359656	1.66

表 5: 普通高斯消元：串行算法与 OpenMp 算法性能对比

图3.2直观地展示了在不同问题规模下的加速比对比。从图中可以看出，OpenMP 技术在普通高斯消元问题上展现出了显著的性能优势。随着问题规模的增大，加速比迅速上升，这表明并行化算法在处理大规模问题时能够显著减少执行时间，提升算法效率。

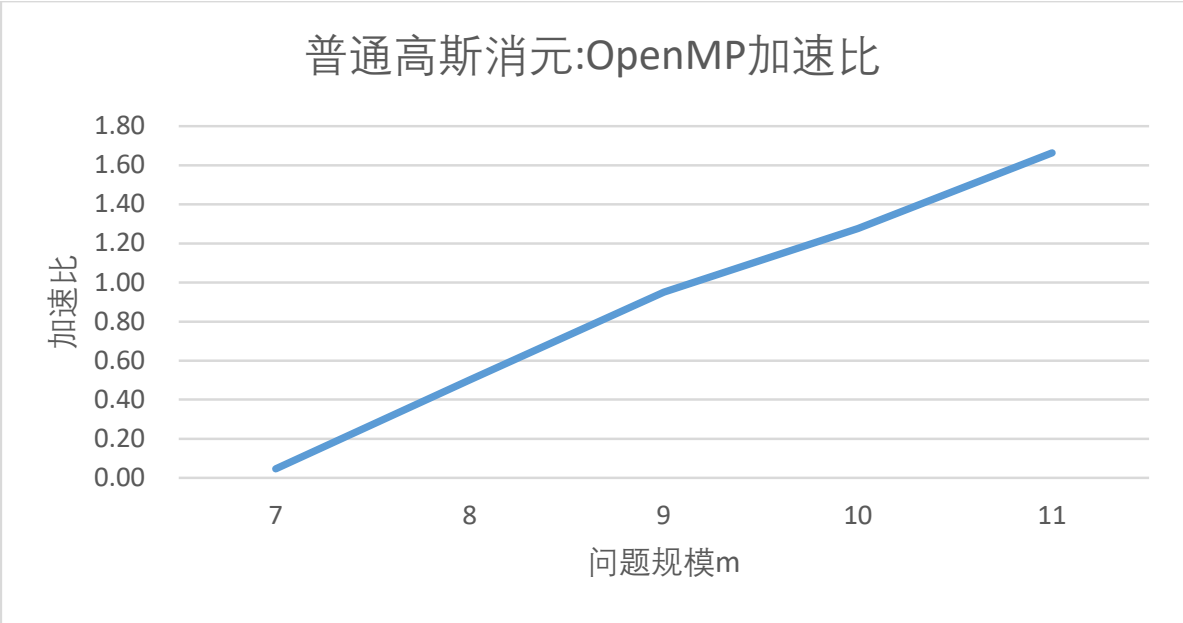

图 3.2: 普通高斯消元：串行算法与 OpenMp 算法加速比

3.3 线程数对多线程算法的影响

在本节中，我们以 test5 为例，探究了线程数对 Pthread 和 OpenMP 两种多线程算法性能的影响。

3.3.1 Pthread 的性能分析

表6和图3.3展示了不同线程数下 Pthread 并行算法的运行时间。

线程数	时间
串行	18895642
1	19935854
2	21717327
3	22430144
4	25245596
5	23134412
6	23379049
7	26317660

表 6: 普通高斯消元：不同线程数下 Pthread 算法性能

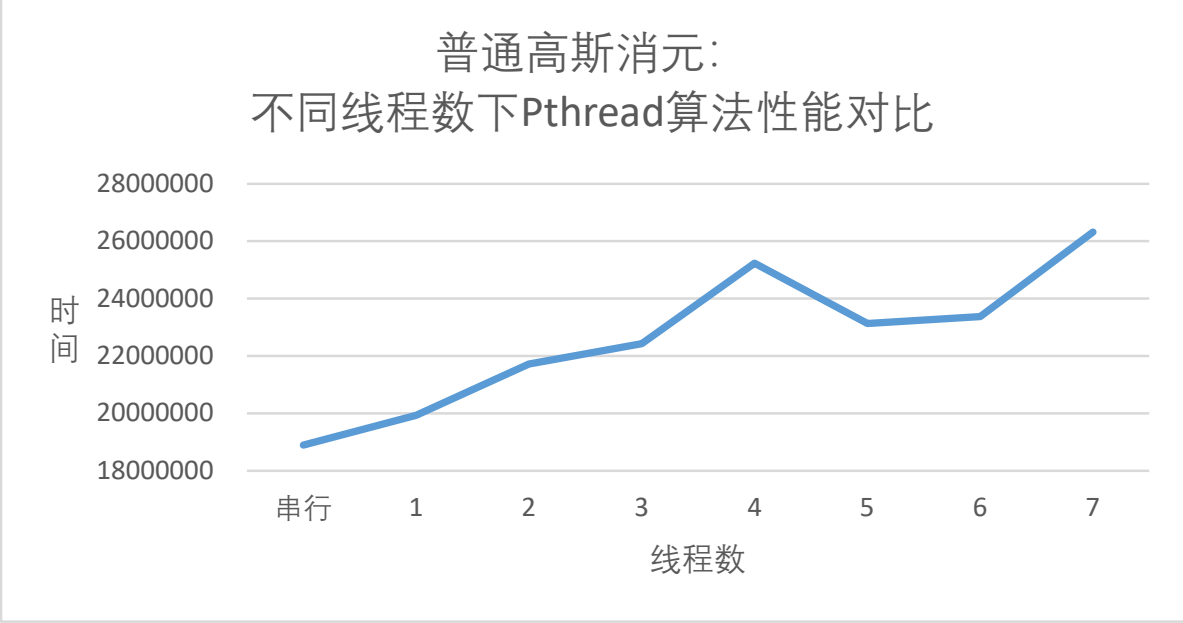

图 3.3: 普通高斯消元：不同线程数下 Pthread 算法性能

分析结果显示，Pthread 技术在本实验条件下未能显著提升算法性能，甚至随着线程数的增加，性能呈现下降趋势。这可能是由于问题规模相对较小，无法充分发挥多线程的优势，以及我们为了保证答案的正确性而采取的严格线程锁操作。

3.3.2 OpenMp 的性能分析

表7和图3.4展示了不同线程数下 OpenMP 并行算法的运行时间。

线程数	时间
串行	18895642
1	22982793
2	10701439
3	7200492
4	5882783
5	4964316
6	4362724
7	3985239

表 7: 普通高斯消元：不同线程数下 OpenMp 算法性能

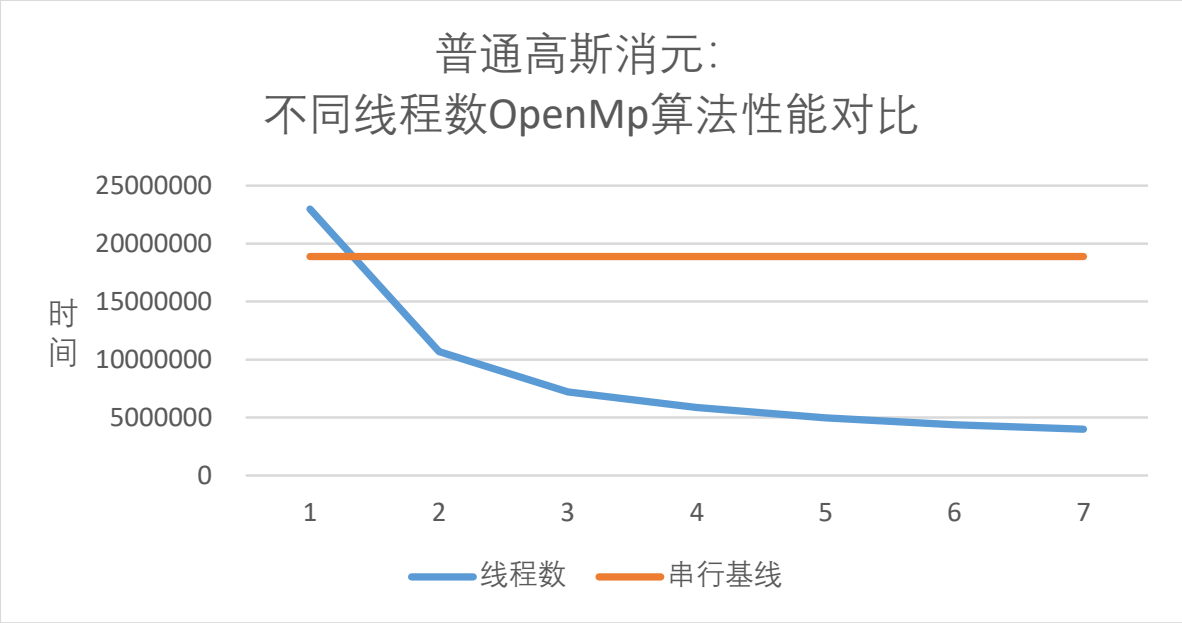

图 3.4: 普通高斯消元：不同线程数下 OpenMp 算法性能

从图表中可以看出，OpenMP 算法能够显著提升算法性能，并且随着线程数的增加，运行时间逐渐减少，表明 OpenMP 算法在处理此类问题时具有良好的可扩展性。这一结果进一步证实了 OpenMP 在并行计算中的优势和有效性。

3.4 总结

在本部分中，我们利用了 Pthread 技术和 OpenMP 技术对普通高斯消元法进行并行化研究，通过分析实验结果，我们得出以下结论：

Pthread 技术在普通高斯消元问题上并未展现出明显的性能提升。尽管随着线程数的增加，加速比有所提升，但始终未能超过 1，表明 Pthread 技术在该问题上并未实现真正的加速效果。这可能是由于问题规模相对较小，未能充分发挥多线程的优势导致的，同时严格的线程锁操作也可能限制了 Pthread 技术的性能。然而，考虑到随着问题规模的增大，加速比有所上升，我们推测在处理超大规模问题时，Pthread 算法可能具备减少执行时间、提升算法效率的潜力。未来研究可进一步探索更为复杂和精细的线程管理机制，以优化 Pthread 技术的性能。

OpenMP 技术在普通高斯消元问题上展现出了显著的性能优势。随着问题规模的增大和线程数的增加，OpenMP 算法的运行时间都迅速减少，加速比迅速上升，这充分说明了 OpenMP 算法在处理大规模问题时能够显著减少执行时间，提升算法效率。

综上所述，虽然 Pthread 技术在当前实验条件下未能显著提升算法性能，但其在处理超大规模问题时的潜力值得进一步挖掘。与此同时，OpenMP 技术以其简单易用和显著的性能提升，成为并行计算中的有力工具。

4 状态压缩 Buchberger 算法

在本部分中，我们将针对状态压缩 Buchberger 算法的 Pthread 并行化和 OpenMP 并行化性能进行深入探讨。通过对比串行算法与并行算法的执行时间，并计算不同问题规模下的加速比，旨在全面评估算法的性能表现。

实验过程中，我们将严格以微秒为单位记录算法的执行时间，确保数据的精确性，从而真实反映算法的执行效率。通过这一系列的实验数据，我们将深入剖析状态压缩 Buchberger 算法在并行化实现下的性能优势与局限性。

4.1 算法设计

4.1.1 Pthread 设计

我们采用 Pthread 技术对前向消元过程进行了并行化优化。以下展示了并行化后的核心代码段，该代码实现了多线程并行执行消元操作，从而提高了消元效率。

前向消元并行化部分

```
1
2  void* onethread_Forward_elimination(void *arg)//前向消元
3  {
4    Parameter para=*(Parameter*)arg;
5    int I=para.I;
6    int begin=para.Id-1;
7    for(int i=begin;i<col2;i=i+MAX_THREADS)
8    {
9      if(D[i]==0) continue;
10     if((B[i][nowrow]&nownum)==0) continue;
11     pthread_mutex_lock(&mutex);
12     for(int j=nowrow;j<=comrow;j++)
13     {
14         B[i][j]=A[I][j]^B[i][j];
15     }
16     pthread_mutex_unlock(&mutex);
17   }
18   pthread_exit(NULL);
19 }
20
21 void Forward_elimination()//前向消元
22 {
23   for(int I=0;I<row;I++)
24   {
25     nowrow=(I>>4);
26     nownum=(1<<(15-(I&15)));
27     if(C[I]==0)
28     {
29       for(int i=0;i<col2;i++)
30       {
31         if(D[i]==0) continue;
```

```
            if((B[i][nowrow]&nownum)==0) continue;
            C[I]=1;
            D[i]=0;
            A[I]=B[i];
            break;
          }
      }
      if(C[I]==0) continue;

      //分配任务,创建线程
      for(int i=1;i<=MAX_THREADS;i++)
      {
        param[i].I=I;
        param[i].Id=i;
        pthread_create(&handles[i],NULL,onethread_Forward_elimination,&param[i]);
      }
      //挂起主线程
      for(int i=1;i<=MAX_THREADS;i++)
      {
        pthread_join(handles[i],NULL);
      }
    }
}
```

在上述代码中，通过 Pthread 相关函数，我们实现了对循环的并行化处理。这使得不同的消元行能够同时进行消元操作，从而实现了多行的并行处理。

4.1.2 OpenMp 设计

我们利用 OpenMp 技术对前向消元过程进行了并行化优化。以下展示了并行化后的核心代码段，该代码利用 OpenMp 的并行循环指令，实现了多线程并行执行消元操作。

前向消元并行化核心部分

```
#pragma omp parallel for
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;j++)
  {
    B[i][j]=A[I][j]^B[i][j];
  }
}
```

在上述代码中，通过 #pragma omp parallel for 指令，我们实现了对循环的并行化处理。这使得不同的消元行能够同时进行消元操作，从而实现了多行的并行处理。

4.2 并行算法的性能分析

4.2.1 Pthread 的性能分析

为了全面评估状态压缩 Buchberger 算法在 Pthread 并行化实现下的性能表现，我们对比了不同问题规模下串行算法与 Pthread 算法的运行时间及加速比。结果如表8所示。

test	串行算法	Pthread 算法	Pthread 加速比
test1	0.00843	2.43620	0.0035
test2	0.23296	9.88797	0.0236
test3	0.49986	14.9038	0.0335
test4	13.8640	49.0073	0.2829
test5	108.568	228.917	0.4743
test6	1576.45	2764.67	0.5702
test7	20806.9	31659.4	0.6572
test8	407289	493052	0.8261

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

图4.5以可视化的方式展示了不同测试案例下 Pthread 算法相较于串行算法的加速比变化。从图中可以看出，尽管在当前的测试规模下，加速比并未超过 1（即并行算法并未真正提升性能），但随着问题规模的增加，加速比呈现出上升趋势。

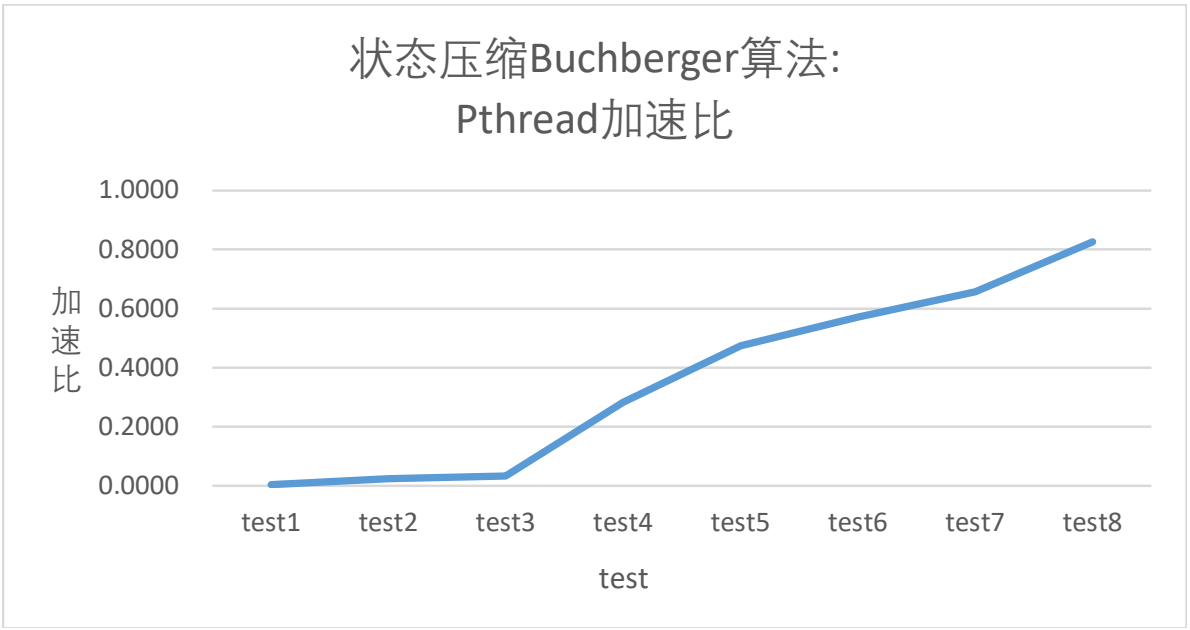

图 4.5: 状态压缩 Buchberger 算法：串行算法与 Pthread 算法的时间性能对比

综上所述，当前 Pthread 并行化并不能提升状态压缩 Buchberger 算法的性能，这可能是由于线程管理开销较大、数据竞争等因素导致的。

随着问题规模的增加，加速比呈现出上升趋势，暗示着在更大的计算规模下，Pthread 并行化有可能带来更好的性能提升。

为了进一步提高算法的性能，未来工作可以探索更精细的线程管理策略、减少数据竞争等方法。

4.2.2 OpenMp 的性能分析

为了深入探究状态压缩 Buchberger 算法在 OpenMp 并行化实现下的性能表现，我们对比了不同问题规模下串行算法与 OpenMp 算法的运行时间及加速比。具体结果如表9所示。

test	串行算法	OpenMp 算法	OpenMp 加速比
test1	0.00843	0.15368	0.05
test2	0.23296	0.26088	0.89
test3	0.49986	0.50023	1.00
test4	13.8640	6.13899	2.26
test5	108.568	43.6386	2.49
test6	1576.45	536.045	2.94
test7	20806.9	6033.08	3.45
test8	407289	136079	2.99
test9	1169586	394484	2.96

表 9: 状态压缩 Buchberger 算法：串行算法与 OpenMp 算法的时间性能对比

图4.6直观地展示了不同测试案例下 OpenMp 算法相较于串行算法的加速比变化。从图中可以观察到，随着问题规模的增加，加速比总体呈现出上升趋势，表明 OpenMp 并行化对于大规模问题具有更好的性能提升效果。

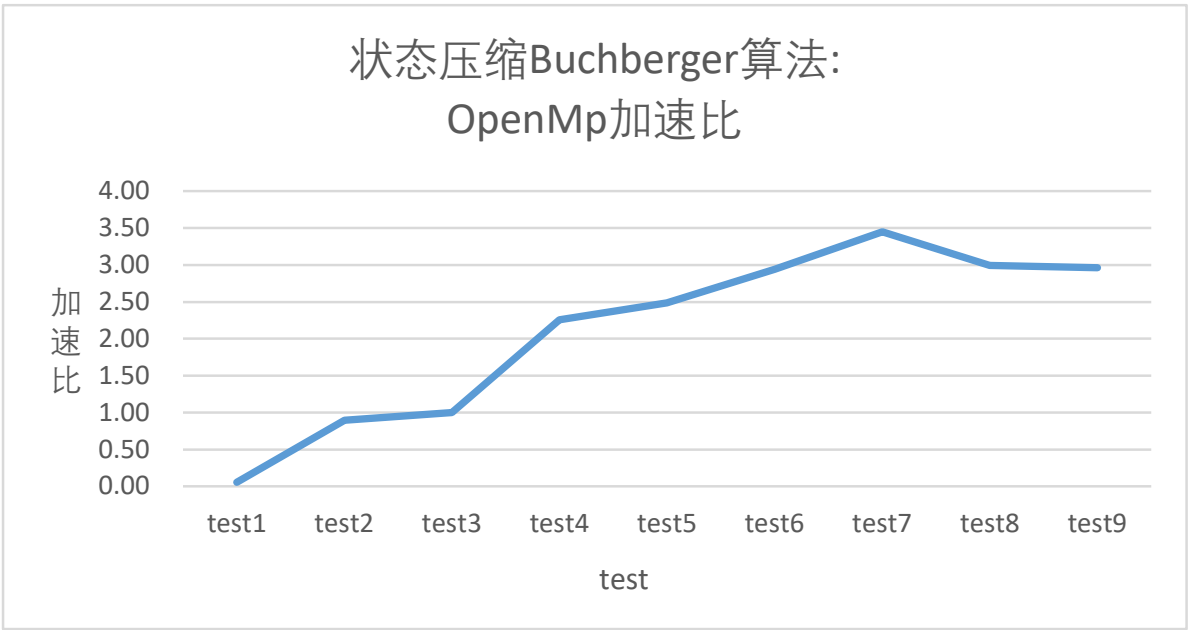

图 4.6: 状态压缩 Buchberger 算法：串行算法与 OpenMp 算法的时间性能对比

对于小规模问题，OpenMp 并行化实现可能由于线程创建、数据分配等开销而未能表现出明显的性能优势。

随着问题规模的增加，OpenMp 并行化实现的性能优势逐渐显现，加速比显著上升，表明 OpenMp 在处理大规模问题时具有更好的并行性能。

为了进一步优化算法性能，可以考虑进一步调整 OpenMp 的并行参数，如线程数、数据划分策略等，以充分发挥多核处理器的并行计算能力。此外，还可以探索其他并行化技术和优化策略，如任务调度、内存管理等，以进一步提升算法的性能。

4.3 线程数对多线程算法的影响

在本节中，我们针对状态压缩 Buchberger 算法，以 test7 为例，详细探究了线程数对 Pthread 和 OpenMP 两种多线程算法性能的影响。

4.3.1 Pthread 的性能分析

为了评估线程数对 Pthread 并行算法性能的影响，我们对比了不同线程数下 Pthread 算法的运行时间，如表10和图4.7所示。

线程数	时间
串行	20806.9
1	21796.2
2	30354.6
3	31146.8
4	31779.4
5	32371.6
6	32180.1
7	32758.2

表 10: 状态压缩 Buchberger 算法：不同线程数下 Pthread 算法性能

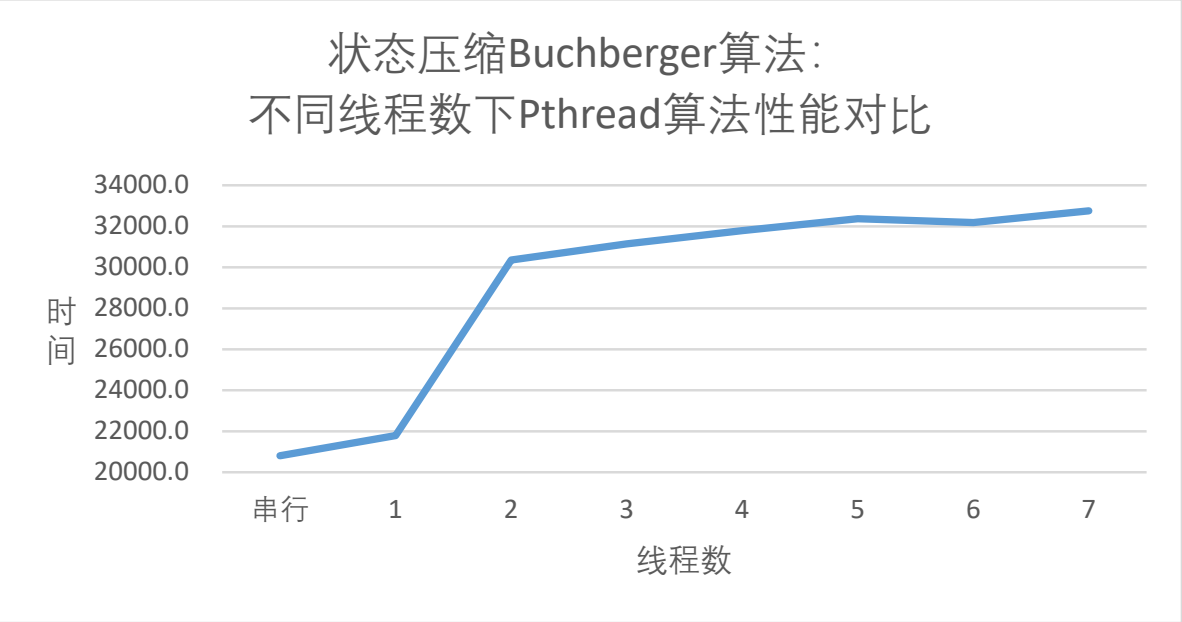

图 4.7: 状态压缩 Buchberger 算法：不同线程数下 Pthread 算法性能

从上述结果可以看出，Pthread 技术在本例中并未能提升状态压缩 Buchberger 算法的性能，反而随着线程数的增加，运行时间逐渐增长，性能呈现下降趋势。这可能是由于 Pthread 的线程管理机制较为复杂，导致线程间的通信和同步开销较大，从而影响了整体性能。

4.3.2 OpenMp 的性能分析

为了与 Pthread 算法进行对比，我们同样测试了不同线程数下 OpenMP 并行算法的运行时间，如表11和图4.8所示。

线程数	时间
串行	20806.9
1	20698.4
2	11252.3
3	7938.01
4	6090.60
5	4911.69
6	4112.03
7	3630.83

表 11: 状态压缩 Buchberger 算法：不同线程数下 OpenMp 算法性能

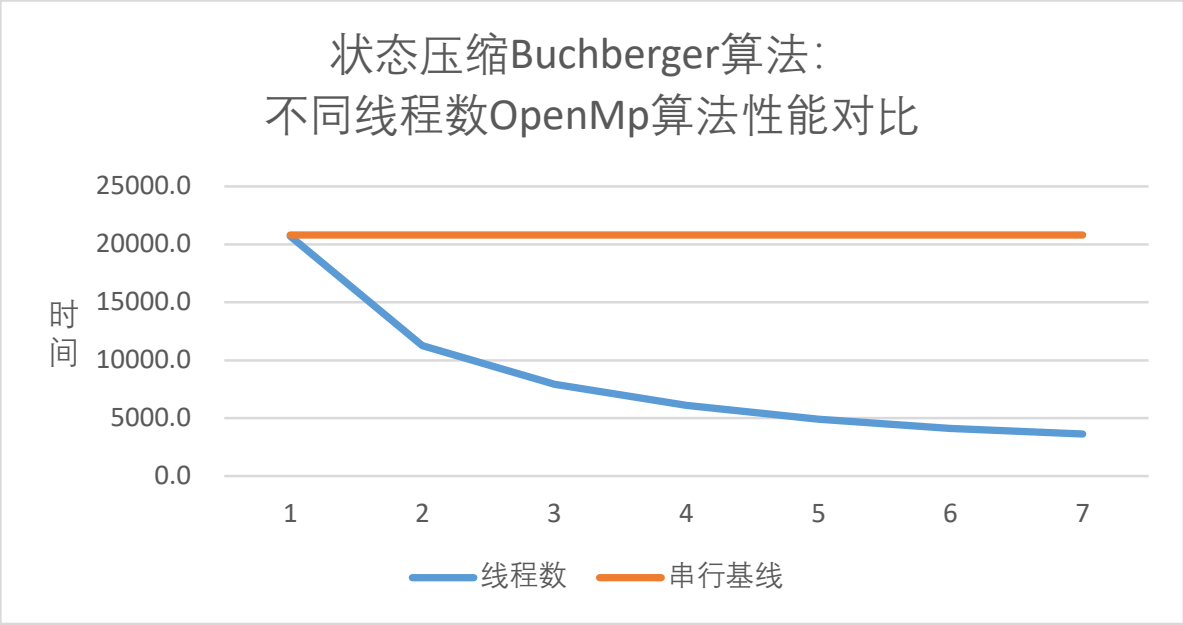

图 4.8: 状态压缩 Buchberger 算法：不同线程数下 OpenMp 算法性能

从表11和图4.8中可以看出，OpenMP 算法在处理状态压缩 Buchberger 算法时表现出显著的性能提升。随着线程数的增加，运行时间逐渐减少，表明 OpenMP 算法在处理此类问题时具有非常良好的可扩展性。这一结果进一步验证了 OpenMP 在状态压缩 Buchberger 算法中的优势和有效性。与 Pthread 相比，OpenMP 通过自动的线程管理和数据分配，减少了线程间的通信和同步开销，从而实现了更好的性能提升。

4.4 profiling 分析

为了深入探索多线程编程技术对状态压缩 Buchberger 算法性能的影响，以及为何其加速比未能达到线程数 4，我们对 pthread 和 OpenMP 技术进行了详细的 profiling 分析。以下分析以 test7 为例，特别关注了 CPI（每条指令的时钟周期数）以及各级缓存命中率的影响。

4.4.1 pthread 的性能分析

表12展示了串行算法与 pthread 算法在时钟周期数、指令数、CPI 以及各级缓存命中率方面的对比。从表中可以看出，尽管 pthread 算法通过多线程实现了并行处理，但其时钟周期数和 CPI 均较串

行算法有所增加。这可能是由于线程间的同步和调度开销所致。然而，pthread 算法的 L2 缓存命中率显著提高，这可能归因于每个线程拥有自己的缓存，这在一定程度上提高了数据访问效率。

	串行算法	Pthread 算法	比值
时钟周期数	54617697997	72008225126	0.76
指令数	164484653041	167599967651	0.98
CPI	0.33	0.43	0.77
L1 cache	99.88%	99.77%	1.00
L2 cache	53.15%	69.50%	0.76
L3 cache	97.42%	95.52%	1.02

表 12: pthread 的 profiling 分析

4.4.2 openmp 的性能分析

表13对比了串行算法与 OpenMP 算法的性能指标。从表中可以看出，OpenMP 算法通过多线程并行执行指令，显著减少了时钟周期数并降低了 CPI，从而提高了算法的执行效率。然而，OpenMP 算法的指令数略高于串行算法，且 L2 缓存命中率相对较低。这可能是由于 OpenMP 在并行处理过程中数据访问模式的变化导致的。这也是其加速比未能达到线程数 4 的重要原因之一。尽管如此，OpenMP 算法仍然取得了较高的加速比，证明了其高效的并行处理能力。

	串行算法	openmp 算法	比值
时钟周期数	54617697997	15144869414	3.61
指令数	164484653041	190551116012	0.86
CPI	0.33	0.08	4.18
L1 cache	99.88%	99.87%	1.00
L2 cache	53.15%	26.97%	1.97
L3 cache	97.42%	96.04%	1.01

表 13: openmp 的 profiling 分析

4.5 OpenMp 与 SIMD 复合算法

在先前的研究中，SIMD 算法和 OpenMp 算法均已被证实能够有效提升状态压缩 Buchberger 算法的性能。为了深入挖掘这两种并行化技术的互补优势，进而提升算法的整体执行效率，本节将探讨将 OpenMp 与 SIMD 技术相结合的方法，并具体应用于状态压缩 Buchberger 算法的前向消元过程中，以实现深度的优化。

4.5.1 算法设计

为了进一步提升状态压缩 Buchberger 算法的执行效率，我们设计了一种复合并行化策略，结合了 OpenMp 的多线程并行处理能力和 ARM NEON 指令集的 SIMD 功能。以下展示了该策略在前向消元过程中的核心实现，它通过 OpenMp 的并行循环指令，实现了多线程环境下的消元操作，并利用 NEON 指令集执行单指令多数据的运算。

前向消元并行化核心部分

```
#pragma omp parallel for
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)
    {
      uint16x8_t A_8=vld1q_u16(&A[I][j]);
      uint16x8_t B_8=vld1q_u16(&B[i][j]);
      B_8=veorq_u16(A_8,B_8);
      vst1q_u16(&B[i][j],B_8);
    }
}
```

在上述代码中，我们首先使用 #pragma omp parallel for 指令来指定循环的并行化。然后，在循环体内部，我们利用 ARM NEON 指令集对每 8 个连续的 16 位无符号整数进行并行处理。

4.5.2 性能分析

为了深入评估状态压缩 Buchberger 算法在 OpenMp-SIMD 复合并行化实现下的性能，我们进行了一系列的实验，对比了串行算法与 OpenMp-SIMD 算法（线程数为 4）在不同问题规模下的运行时间及加速比。具体结果如表14所示。

test	串行算法	OpenMp-SIMD 算法	OpenMp-SIMD 加速比
test1	0.00843	0.16929	0.05
test2	0.23296	0.31099	0.75
test3	0.49986	0.51791	0.97
test4	13.8640	3.81271	3.64
test5	108.568	20.9309	5.19
test6	1576.45	227.803	6.92
test7	20806.9	2556.80	8.14
test8	407289	55527.5	7.33
test9	1169586	155651	7.51

表 14: 状态压缩 Buchberger 算法：串行算法与 OpenMp-SIMD 算法的时间性能对比

从表14中可以看出，随着问题规模的增大，OpenMp-SIMD 算法相较于串行算法展现出了显著的加速效果。为了进一步分析 OpenMp 和 SIMD 算法的结合效果，我们定义 OpenMp-SIMD 算法的理论加速比为 OpenMp 算法的加速比与 SIMD 算法的加速比的乘积。我们计算了 OpenMp-SIMD 算法的理论加速比和实际加速比，并进行了对比，如表15所示。

test	OpenMp 加速比	simd 加速比	OpenMp-simd 理论加速比	OpenMp-simd 实际加速比
test1	0.05	1.07	0.06	0.05
test2	0.89	1.28	1.14	0.75
test3	1.00	1.49	1.49	0.97
test4	2.26	1.91	4.31	3.64
test5	2.49	2.27	5.65	5.19
test6	2.94	2.21	6.50	6.92
test7	3.45	2.24	7.73	8.14
test8	2.99	2.41	7.21	7.33

表 15: 状态压缩 Buchberger 算法：OpenMp-simd 算法理论与实际加速比对比

图4.9直观地展示了不同测试案例下 OpenMp-SIMD 算法理论与实际加速比的对比。从图中可以

看出，实际加速比与理论加速比在大部分情况下高度接近，这表明 OpenMp 和 SIMD 技术的结合在状态压缩 Buchberger 算法中发挥了良好的协同作用。特别是在大规模问题上，实际加速比略高于理论加速比，这可能是由于多核处理器上的线程调度和缓存利用等因素带来的额外性能提升。

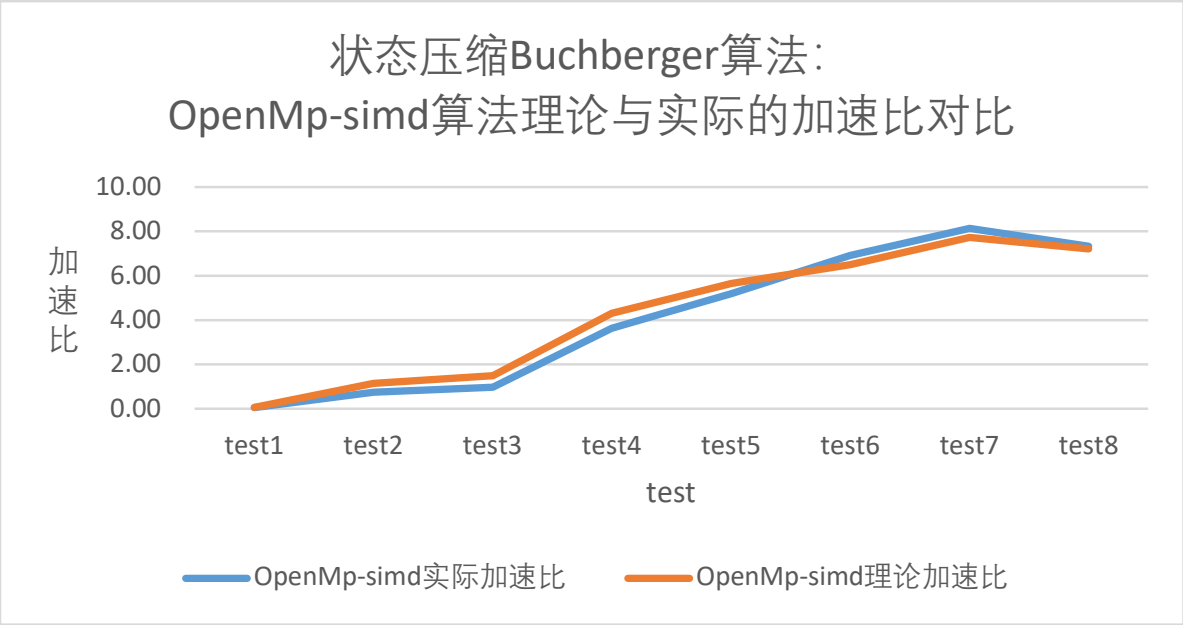

图 4.9: 状态压缩 Buchberger 算法：OpenMp-simd 算法理论与实际的时间性能对比

4.6 总结

在本节中，我们对状态压缩 Buchberger 算法的 Pthread 并行化和 OpenMP 并行化性能进行了系统的研究和实验分析。通过对比串行算法与并行算法的执行时间，并计算不同问题规模及线程数下的加速比，我们全面评估了算法的性能表现。

OpenMP 并行化优势 实验结果表明，OpenMP 并行化在实现状态压缩 Buchberger 算法时展现出显著的性能优势。随着问题规模的增大，OpenMP 并行化的加速比逐渐上升，特别是在处理大规模问题时，其性能提升效果尤为显著。OpenMP 的自动并行化和线程调度机制在处理复杂算法时具有明显优势，能够减少处理器的空闲时间，提高 CPU 的利用率，从而显著缩短算法的执行时间。

Pthread 并行化挑战 相比之下，Pthread 并行化在当前测试环境下并未能提升算法性能，反而随着线程数的增加，性能呈现下降趋势。这主要归因于 Pthread 线程管理机制的复杂性以及线程间通信和同步的开销较大。此外，Pthread 需要程序员手动管理线程和同步，增加了编程的复杂性和出错的可能性。在数据共享和竞争条件处理方面，Pthread 也面临一定的挑战。

Profiling 分析 通过 profiling 分析，我们进一步揭示了多线程并行化对状态压缩 Buchberger 算法性能的影响。OpenMP 并行化在处理大规模数据集时展现出较高的效率，而 Pthread 并行化则在处理器利用率和线程同步开销方面表现不佳。

OpenMP 与 SIMD 复合算法探索 为了进一步提高算法性能，我们探索了 OpenMP 与 SIMD 复合算法的实现。SIMD（单指令多数据）技术允许处理器同时对多个数据项执行相同的操作，从而提高了

数据处理的吞吐量。结合 OpenMP 的多线程并行处理能力，我们成功地实现了更高效的算法执行。实验结果表明，OpenMP 与 SIMD 复合算法在处理大规模数据集时取得了显著的性能提升。

结论 综上所述，OpenMP 并行化在实现状态压缩 Buchberger 算法时表现出良好的性能优势，而 OpenMP 与 SIMD 复合算法则进一步提高了算法的性能。这些并行化策略对于状态压缩 Buchberger 算法具有重要的实际应用价值，特别是在处理大规模问题时。未来工作将探索更多优化策略，以进一步提升算法的性能和效率。

5 总结

本文针对普通高斯消元法和状态压缩 Buchberger 算法，利用 Pthread 和 OpenMP 两种并行编程技术进行了深入的实验研究和性能分析。实验结果表明，不同算法和并行化策略在处理不同规模问题时展现出各异的性能特点。

对于普通高斯消元法，Pthread 并行化技术并未在实验中展现出显著的性能提升。尽管随着问题规模的增大，加速比有所提升，但始终未能超过 1，表明 Pthread 技术在该算法上并未实现真正的加速效果。这可能是由于实验中使用的问题规模相对较小，未能充分发挥多线程的优势。同时，严格的线程锁操作也可能限制了 Pthread 技术的性能。然而，考虑到随着问题规模的增大，加速比有所上升，我们推测在处理超大规模问题时，Pthread 算法可能具备减少执行时间、提升算法效率的潜力。因此，未来研究可进一步探索更为复杂和精细的线程管理机制，以优化 Pthread 技术的性能。

相比之下，OpenMP 技术在普通高斯消元法上展现出了显著的性能优势。随着问题规模的增大和线程数的增加，OpenMP 算法的运行时间显著减少，加速比迅速上升。这充分说明了 OpenMP 算法在处理大规模问题时能够显著减少执行时间，提升算法效率。OpenMP 的自动并行化和线程调度机制在处理复杂算法时具有明显优势，能够减少处理器的空闲时间，提高 CPU 的利用率。

在状态压缩 Buchberger 算法上，OpenMP 并行化同样表现出良好的性能优势。随着问题规模的增大，OpenMP 并行化的加速比逐渐上升，尤其是在处理大规模问题时，其性能提升效果尤为显著。通过 profiling 分析，我们进一步验证了 OpenMP 在处理大规模数据集时的高效率。此外，我们还探索了 OpenMP 与 SIMD 复合算法的实现，并发现这种复合并行化策略能够进一步提高算法的性能，特别是在处理大规模数据集时取得了显著的性能提升。

相比之下，Pthread 并行化在状态压缩 Buchberger 算法中并未能提升算法性能，反而随着线程数的增加，性能呈现下降趋势。这主要归因于 Pthread 线程管理机制的复杂性以及线程间通信和同步的开销较大。此外，Pthread 需要程序员手动管理线程和同步，增加了编程的复杂性和出错的可能性。

综上所述，OpenMP 并行化技术在普通高斯消元法和状态压缩 Buchberger 算法上均展现出显著的性能优势，特别是在处理大规模问题时，其性能提升效果尤为明显。而 Pthread 并行化技术在当前实验条件下并未能显著提升算法性能，但在处理超大规模问题时可能具备潜力。未来研究可进一步探索更为精细的线程管理机制和优化策略，以进一步提升 Pthread 技术的性能，并探索更多并行化技术和优化策略，以满足不同算法在不同应用场景下的性能需求。