南開大學
Nankai University

计算机学院

并行程序设计实验报告

自选题目：基于 COO 格式的稀疏矩阵乘法 MPI+ 优化

姓名：孟笑朵

学号：2010349

专业：计算机科学与技术

2022 年 6 月 17 日

Abstract

在本次实验中，利用常规的稀疏矩阵存储方式 COO 进行存储，实现了稀疏矩阵乘法 SpMM 和稀疏矩阵向量乘法 SpMV 的 MPI 优化并行算法，并结合 SIMD 算法,pthread 算法,openMP 算法进行进一步优化，分别实现 window 平台平台，金山云 x86 平台和鲲鹏 arm 平台的并行化加速，并利用性能测试与分析工具分析其相比于平凡算法的性能提升。

关键字： SpMV，SpMM，MPI

目录

1 性能测试环境

编程语言：C++
编译器：TDM-GCC
Windows10 操作系统:

- 中心处理器：AMD Ryzen 7 5800H
- 硬盘：512GB
- 内存：16GB

测试得到本机一共可以开启 16 个线程
华为鲲鹏服务器:

- 中心处理器: Linux master 4.14.0-115.el7a.0.1.aarch64
- 内存：195907MB

其中每个共有 96 个 CPU，每个 CPU 支持一个线程，一个 CPU 有 48 个核, 一共可以开启两个节点
金山云 x86 平台

- 中心处理器：Intel Xeon Processor (Cascadelake)
- 内存: 3716MB

一共可以开启 32 个节点, 每个节点可以开启 4 个线程.

2 自选题介绍

本文是作为同杨鑫合作的《WMD 算法的并行化优化》的一个子问题的解决方案，该问题聚焦于 WMD 算法中最终求解的 Sinkhorn-Knopp 算法的并行优化，算法中涉及到多次稀疏矩阵与稠密向量矩阵相乘，因此，实现稀疏矩阵乘法的并行化十分有必要。不仅在本算法中，稀疏矩阵/张量在科学计算、数据分析、机器学习等应用中也十分常见，而稀疏矩阵的间接内存访问模式又给代码优化带来了巨大挑战，实现一种高效的稀疏矩阵乘法算法意义重大。

在《基于 COO 格式的稀疏矩阵乘法的 SIMD 编程》中，我们已经详细阐述了关于稀疏矩阵向量乘法的一些概念,并对稀疏矩阵压缩格式进行简单的介绍。在《基于 COO 格式的稀疏矩阵乘法的 pThread 编程》中，我们进行了稀疏矩阵向量乘的 pThread 并行优化并进行了性能测试，并对 HYB(HYBrid) 格式表示的稀疏矩阵进行了简单介绍。在《基于 COO 格式的稀疏矩阵乘法的 openMP 编程》中，我们对 COO 格式的稀疏矩阵乘法进行 openMP 编程的并行化优化，并对比分析了 pthread 并行化优化效果，同时结合 SIMD 进行进一步的优化。在《基于 COO 格式的稀疏矩阵乘法的 GPU 优化》中我们对稀疏矩阵乘法进行 CUDA 编程, 并在不同平台上测试对比了其性能。

本实验将继续基于 COO 格式的稀疏矩阵乘法并行化优化，本实验中继续将稀疏矩阵乘法划分为两类：一类是稀疏矩阵稠密向量相乘（SpMV），一类是稀疏矩阵稠密矩阵相乘（SpMM），将这两类问题分别实现 x86 平台和鲲鹏 arm 平台的 MPI 并行化加速，并比较分析其相比于平凡算法的性能提升效果。

3 并行算法实现与性能分析

3.1 SpMV 串行算法与并行分析

基于 COO 的稀疏矩阵格式与稠密矩阵相乘的算法伪代码如下：

$$\text{for } l = 0 \text{ to } k-1 \text{ do}$$
$$y[\text{ rowind }[l]] \leftarrow \text{y[rowind }[l]] + \text{val}[l] \cdot \text{x[colind}[l]]$$

在 SIMD 中，我们对这种算法进行了访存分析，发现这种算法至少要进行五次 memory load 才能进行一次运算，在 pThread 编程中，我们优化了 COO 的稀疏矩阵表示格式，并说明了对外层的多线程优化要明显优于对内层的多线程并行化。

COO 存储改进格式

```
class COOMatrix
{
    float* value,//存储非零元素的值
    int* row,//存储非零元素的行下标
    int* col,//存储非零元素的列下标
    int* index//存储每行第一个非零元素在row中的下标
};
```

在 pthread 和 openMP 的并行化优化中，我们运用以上改进的存储格式对平凡算法做了修改，并在此基础上进行了并行化优化，算法伪代码如下：

```
int i,j;
for(i=0;i<nozerorows;i++)
{
    for(j=index[i];j<index[i+1];j++)
    {
        yy[row[j]]+=value[j]*vec[col[j]];
    }
}
```

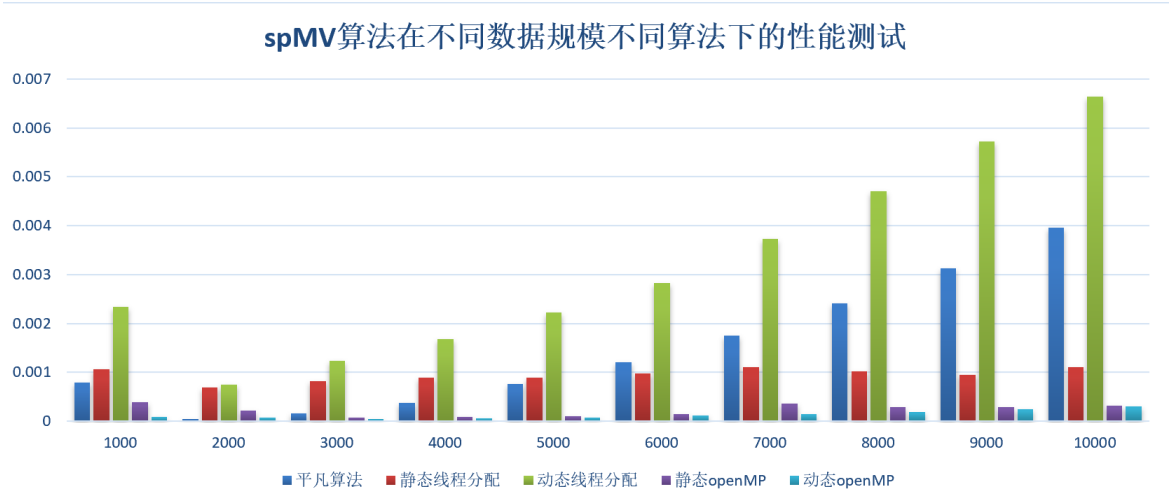
3.2 SpMV 的 MPI+ 并行化算法

在本次实验中，我们继续基于以上的平凡算法进行优化，很明显的一个思路是将数据进行按行划分，将不同数据分配给不同的进程，最先想到的划分方式是: 比如有 100 行的数据, 要分配给 4 个进程, 划分方式可以是第一个进程 0-24, 第二个 25-49, 第三个 50-74, 第四个 75-100, 但是这种划分方式实现起来较为繁琐, 而且对于不能恰好以整数划分的行数处理起来较为麻烦, 我们使用类似等差数列间隔的划分方式, 自然将数据划分为 4 大块, 在 x86 平台下算法优化代码如下：

```
void coo_multiply_vector_mpi(int argc, char* argv[]){
    int myid, numprocs;
```

```
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    for(int i=myid;i<nozerorows;i+=numprocs)
    {
        for(int j=index[i];j<index[i+1];j++)
        {
            yy[row[j]]+=value[j]*vec[col[j]];
        }
    }
    MPI_Finalize();
}
```

在这种算法的基础上我们可以进行 pThread 或者 openMP 多线程编程改进, 在 openMP 的实验中我们发现对比 pthread 来讲 openMP 的优化效果更好，在数据规模较小的情况下，多线程的优化效果并不佳，而 openMP 的优化效果要明显优于平凡算法和 pThread 优化算法, 因此在这里采用 openMP 多线程对 MPI 并行化程序作进一步的优化:



MPI 并行算法结合 openMP 算法的优化算法代码如下:

```
void coo_multiply_vector_mpi(int argc, char* argv[]){
    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    int i,j;
    #pragma omp parallel for num_threads(OMP_NUM_THREADS),private(i, j)
    for(i=myid;i<nozerorows;i+=numprocs)
    {
        for(j=index[i];j<index[i+1];j++)
```

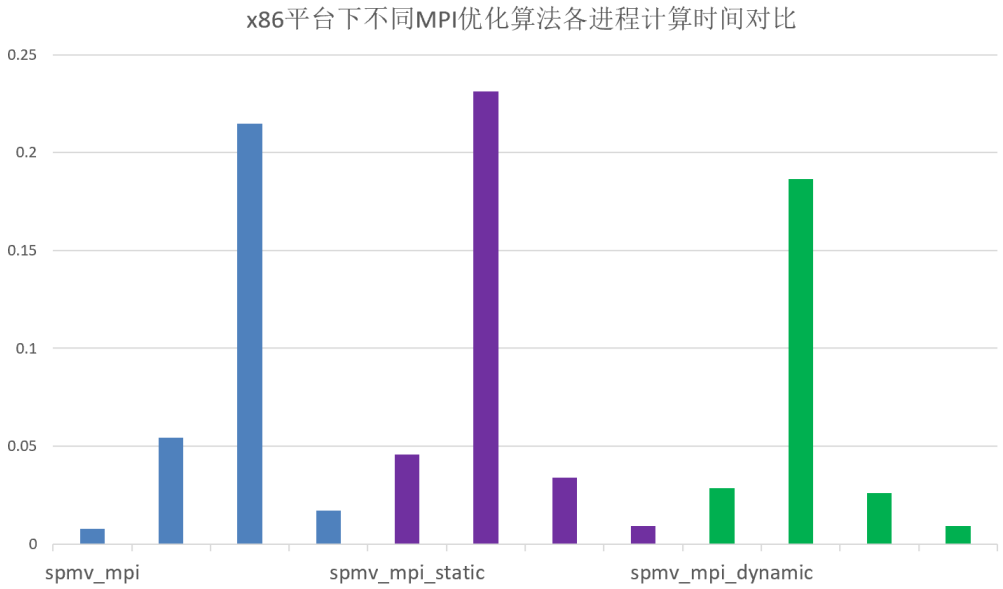
```
        {
            yy[row[j]]+=value[j]*vec[col[j]];
        }
    }
    MPI_Finalize();
}
```

openMP 为我们提供了很方便的 API，使得我们很容易进行静态动态线程分配，这里实现 guided 动态调整的动态线程分配算法，运用 openMP 的动态线程分配 API 来解决其中出现的负载不均问题:

```
void coo_multiply_vector_mpi(int argc, char* argv[]){
    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    int i,j;
    #pragma omp parallel num_threads(OMP_NUM_THREADS),private(i, j)
    //#pragma omp for schedule(static, nozerorows/OMP_NUM_THREADS)dynamic, 50
    #pragma omp for schedule(guided)
    for(i=myid;i<nozerorows;i+=numprocs)
    {
        for(j=index[i];j<index[i+1];j++)
        {
            yy[row[j]]+=value[j]*vec[col[j]];
        }
    }
    MPI_Finalize();
}
```

3.3 SpMV 的 MPI 并行化算法的性能测试

但是在 MPI 的多进程多线程并行化中，我们还是发现无论是否采用 openMP 进行动静态线程划分都会出现负载不均的情况，我们采用的矩阵数据规模为 10000，开启的节点数目为 4，总进程为 4，具体如下图所示:



同时对比平凡算法, 我们发现尽管 MPI 进行优化, 但是其中的时间花费要远远大于平凡算法的时间损耗, 开启一个进程耗费的资源远远大于计算的耗费, 如下表所示:

算法	时间/s
serial	0.000366
pthread_static	0.000536
pthread_dynamic	0.001059
openMP_static	0.000442
openMP_dynamic	0.000428
serial	0.00037
pthread_static	0.000361
pthread_dynamic	0.001048
openMP_static	0.000444
openMP_dynamic	0.000454
serial	0.000379
pthread_static	0.000381
pthread_dynamic	0.001002
openMP_static	0.000451
openMP_dynamic	0.000446
serial	0.000366
pthread_static	0.000376
pthread_dynamic	0.000993
openMP_static	0.000456
openMP_dynamic	0.000441
spmv_mpi	0.214783
spmv_mpi_static	0.231385
spmv_mpi_dynamic	0.186639

由于 SpMV 算法的 SIMD 并行化效果并不佳，在本节中没有对 MPI 和 SIMD 并行化进行结合，下面主要来看一下 SpMM 算法的并行化优化：

3.4 SpMM 串行算法与并行分析

在 SIMD 实验中，我们进行简单的 Cache 优化所得到稀疏矩阵相乘算法伪代码如下：

```
void coo_multiply_matrix_serial(){
    for (int i=0;i<nonzeros;i++)
        for(int k=0;k<n;k++)
            c[row[i]][k] += value[i] * b[col[i]][k];
}
```

对于稀疏矩阵乘法，我们采取的 MPI 并行化思路与 openMP 和 pThread 中多线程并行化思路类似，在稀疏矩阵乘法的外层循环进行多进程循环划分，划分的思路和前面对于 spMV 的划分思路类似，都是采用类似等差数列的方式将计算量进行划分，具体编程如下：

```
void coo_multiply_matrix_mpi(int argc, char* argv[]){
    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    for (int i=myid;i<nonzeros;i+=numprocs)
    {
        for(int k=0;k<n;k++)
            mat_res2[row[i]][k] += value[i] * mat_nonsparse[col[i]][k];
    }
    MPI_Finalize();
}
```

3.5 spMM 的 MPI+openMP 并行化优化

同样的，在每一个进程内部，我们可以继续进行多线程的任务划分，这里与 spMV 的并行化思路类似，都是采用结合 openMP 进行多进程多线程并行化，算法代码如下：

```
void coo_multiply_matrix_mpi(int argc, char* argv[]){
    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    int i,k;
    #pragma omp parallel for num_threads(OMP_NUM_THREADS),private(i, k)
    for (i=myid;i<nonzeros;i+=numprocs)
    {
        for(k=0;k<n;k++)
```

```
            mat_res2[row[i]][k] += value[i] * mat_nonsparse[col[i]][k];
    }
    MPI_Finalize();
}
```

同样的, 我们也可以进行动态线程分配, 这里就不再赘述了.

3.6 SpMM 的 MPI+openMP+SIMD 并行化优化

进一步地，我们看到我们只对外层循环进行了任务划分，而并没有对内层循环进行任务划分，对于内层循环，正好容易利用内层循环 cache 空间访问的特性对内层循环进行 SIMD 并行化优化，并行化优化的思路区分 x86 和 arm 平台有如下几种实现范式:

1. openMP 静态线程分配 +sse 并行化
2. openMP 静态线程分配 +avx 并行化
3. openMP 动态线程分配 +sse 并行化
4. openMP 动态线程分配 +avx 并行化
5. openMP 静态线程分配 +neno 并行化
6. openMP 动态线程分配 +neno 并行化

我们在这里列举 x86 平台的两种算法和 ARM 平台的两种算法, 代码如下:

```
///实现 MPI+openMP+dynamic+sse
void coo_multiply_matrix_mpi_dynamic_sse(int argc, char* argv[]){
    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    __m128 t1,t2,t3,sum;
    int choice = n % 4;
    int i,k;
    #pragma omp parallel num_threads(OMP_NUM_THREADS),private(i, k,t1,t2,t3,sum)
    //#pragma omp for schedule(static, nozerorows/OMP_NUM_THREADS)dynamic, 50
    #pragma omp for schedule(guided)
    for (i=myid;i<nonzeros;i+=numprocs)
    {
        for(k=0;k<n-choice;k+=4)
            {
                t1=_mm_load_ps(mat_nonsparse[col[i]]+k);
                sum = _mm_setzero_ps();
                t3 = _mm_set_ps1(value[i]);
                t2=_mm_load_ps(mat_res1[row[i]]+k);
                sum = _mm_mul_ps(t3,t1);
                t2=_mm_add_ps(t2,sum);
```

```
                _mm_store_ps(mat_res1[row[i]]+k,t2);
            }
        for(k=n-choice;k < n;k++){
            mat_res1[row[i]][k] += value[i] * mat_nonsparse[col[i]][k];
        }
    }
    MPI_Finalize();
}
///实现 MPI+openMP+dynamic+avx
void coo_multiply_matrix_mpi_dynamic_avx(int argc, char* argv[]){
    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    __m256 t1, t2, t3, sum;
    int choice = n % 8;//对齐
    int i,k;
    #pragma omp parallel num_threads(OMP_NUM_THREADS),private(i, k)
    //#pragma omp for schedule(static, nozerorows/OMP_NUM_THREADS)dynamic, 50
    #pragma omp for schedule(guided)
    for (i=myid;i<nonzeros;i+=numprocs)
    {
        for(k=0;k<n-choice;k+=8)
        {
            sum = _mm256_setzero_ps();
            t1=_mm256_loadu_ps(mat_nonsparse[col[i]]+k);
            t3 = _mm256_set1_ps(value[i]);
            t2=_mm256_loadu_ps(mat_res1[row[i]]+k);//将值 load 进向量
            sum = _mm256_mul_ps(t3,t1);//对位相乘
            t2=_mm256_add_ps(t2,sum);//对位相加
            _mm256_storeu_ps(mat_res1[row[i]]+k,t2);//对位存储
        }//处理剩余元素
        for(k=n-choice;k < n;k++){
            mat_res1[row[i]][k] += value[i] * mat_nonsparse[col[i]][k];
        }
    }
    MPI_Finalize();
}
```

类似地，可以在 ARM 鲲鹏平台上实现 MPI 结合 openMP 结合 NEON 的并行化算法，代码如下:

```
///实现 MPI+openMP+static+neon
```

```
void coo_multiply_matrix_mpi_static_neon(int argc, char* argv[]){
    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    float32x4_t t1,t2,t3,sum;
    int choice = n % 4;
    int i,k;
    #pragma omp parallel for num_threads(OMP_NUM_THREADS),private(i,
    k,t1,t2,t3,sum),shared(choice,mat_nonsparse,mat_res1,value,row,col)
    for (i=myid;i<nonzeros;i+=numprocs)
    {
        for(k=0;k<n-choice;k+=4)
        {
            sum=vdupq_n_f32(0.f);
            t1=vld1q_f32(mat_nonsparse[col[i]]+k);
            t2=vld1q_f32(mat_res1[row[i]]+k);
            t3 = vdupq_n_f32(value[i]);
            sum = vmulq_f32(t3,t1);
            t2= vaddq_f32(t2,sum);
            vst1q_f32(mat_res1[row[i]]+k,t2);
        }
        for(k=n-choice;k < n;k++){
            mat_res1[row[i]][k] += value[i] * mat_nonsparse[col[i]][k];
        }
    }
    MPI_Finalize();
}
///实现 MPI+openMP+dynamic+neon
void coo_multiply_matrix_mpi_dynamic_neon(int argc, char* argv[]){
    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    float32x4_t t1,t2,t3,sum;
    int choice = n % 4;
    int i,k;
    #pragma omp parallel num_threads(OMP_NUM_THREADS),private(i,
    k,t1,t2,t3,sum),shared(choice,mat_nonsparse,mat_res1,value,col,row)
    //#pragma omp for schedule(static, nozerorows/OMP_NUM_THREADS)dynamic, 50
    #pragma omp for schedule(guided)
    for (i=myid;i<nonzeros;i+=numprocs)
```

```
    {
        for(k=0;k<n-choice;k+=4)
        {
            sum=vdupq_n_f32(0.f);
            t1=vld1q_f32(mat_nonsparse[col[i]]+k);
            t2=vld1q_f32(mat_res1[row[i]]+k);
            t3 = vdupq_n_f32(value[i]);
            sum = vmulq_f32(t3,t1);
            t2= vaddq_f32(t2,sum);
            vst1q_f32(mat_res1[row[i]]+k,t2);
        }
        for(k=n-choice;k < n;k++){
            mat_res1[row[i]][k] += value[i] * mat_nonsparse[col[i]][k];
        }
    }
    MPI_Finalize();
}
```

3.7 SpMM 并行算法性能分析

3.7.1 MPI 并行算法不同进程的负载分析

首先, 我们针对不同的 MPI 优化算法在 x86 平台上进行程序性能分析, 统计了各种 MPI 并行算法不同进程的程序运行时间, 发现还是存在负载不均的情况, 如下图所示:

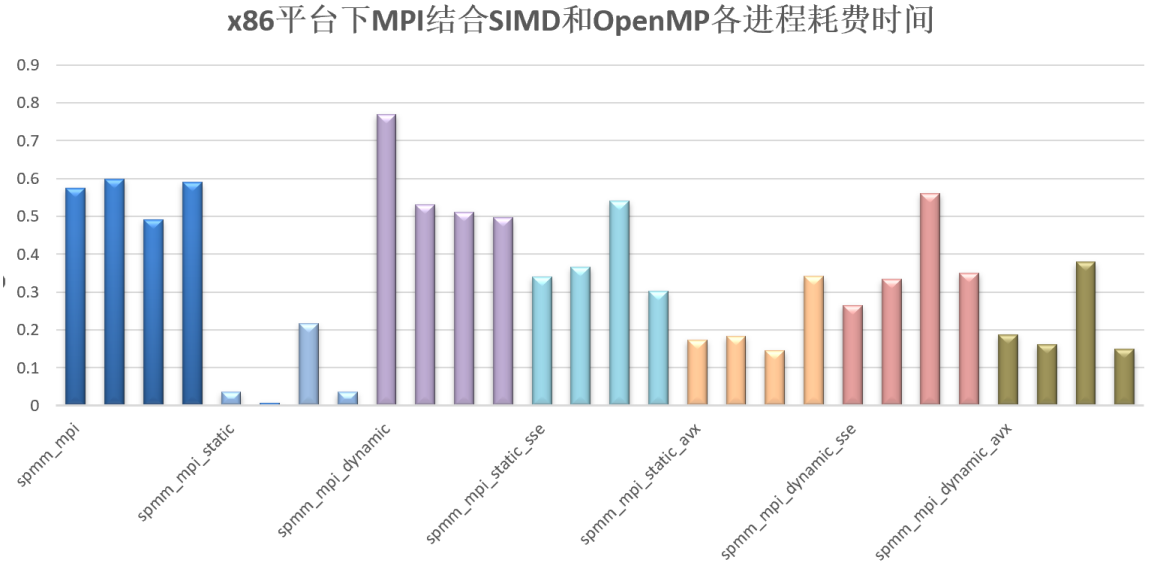


图 3.1: 金山云 x86 平台下不同 MPI 优化算法各进程的运行时间

图中测试数据规模为 4096, 矩阵稀疏度为 0.005, 进程数目为 4, 每个进程的线程数目为 4. 分析图我们发现, 原本的 MPI 中各进程的运行时间相差不大, 大致为原平凡算法运行时间的 1/4, 但是当结

合 openMP 多线程和 SIMD 时, 各进程的负载出现了不均衡的情况, 对于 ARM 平台下, 我们发现了同样的问题:

算法	进程编号	时间/s
spmm_mpi	1	0.555379
	2	0.255831
spmm_mpi_static	1	0.538757
	2	0.275455
spmm_mpi_dynamic	1	0.510719
	2	0.277109
spmm_mpi_static_neon	1	0.510025
	2	0.264334
spmm_mpi_dynamic_neon	1	0.517194
	2	0.265159

3.7.2 MPI 并行算法与其他算法的对比分析

我们分别在金山云 x86 平台, 华为鲲鹏 arm 平台上进行了程序性能测试, 测试了平凡算法, SIMD 的各种算法, pthread 各种算法, openMP 各种算法和 MPI 的各种算法的运行的性能, 数据规模为 4096, 进程数目为 4, 线程数目为 4, 测试结果如下图所示:

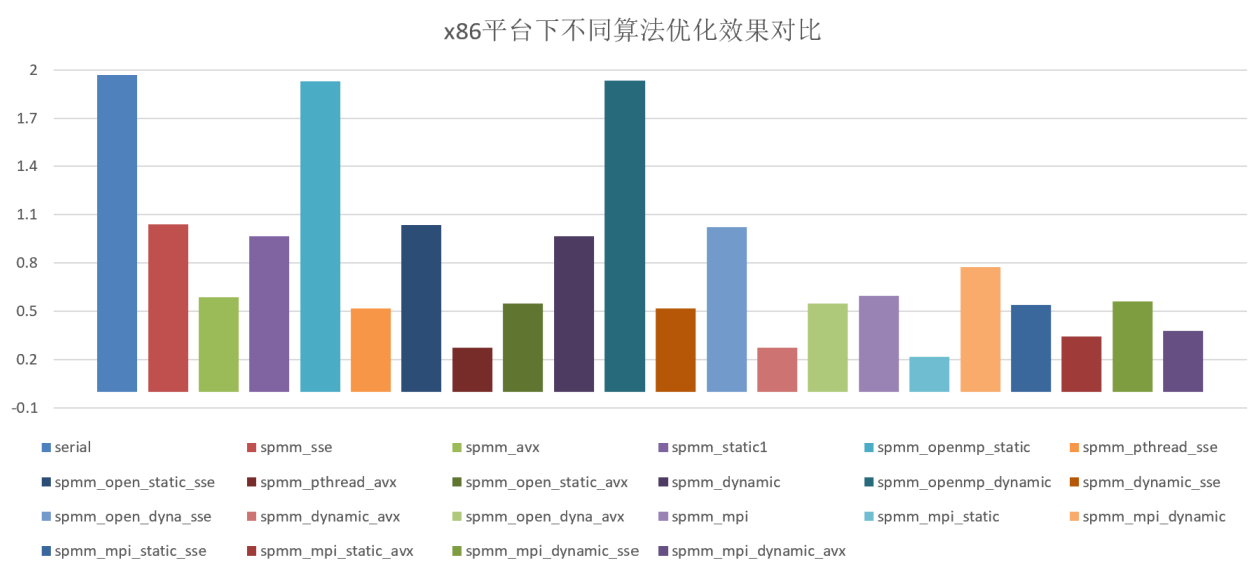


图 3.2: x86 平台上各种算法的程序性能对比

可以发现对在数据规模为 4096 的情况下, 在 x86 平台下各种算法中动静态 pthread 线程分配性能最优, 达到了平凡算法 10 倍的效率, 而 MPI 的各种算法基于中游水平.

ARM 鲲鹏平台上的程序性能分析如下表所示, 其中矩阵的规模为 4096,openMP 和 pthread 的线程数目均为 4.MPI 的进程数目均为 4.

算法	时间/s
serial	0.48794
spmm_neon	0.48128
spmm_static	0.143972
spmm_openmp_static	0.487886
spmm_dynamic	0.136249
spmm_openmp_dynamic	0.545325
spMM_static_neon	0.135722
spmm_open_static_neon	0.473895
spmm_dynamic_neon	0.134561
spmm_open_dyna_neon	0.445809
spmm_mpi	0.555379
spmm_mpi_static	0.538757
spmm_mpi_dynamic	0.510719
spmm_mpi_static_neon	0.510025
spmm_mpi_dynamic_neon	0.517194

发现 MPI 算法的几次优化花费时间都差不多, 在华为鲲鹏平台上面的 MPI 优化算法结果可能存在一定问题, 进行具体的性能分析. 发现进程在主节点和子节点的运行被中止, 最后结果的时间并不准确.

3.7.3 不同数据规模下不同算法在 x86 平台下的性能分析

最后, 我们结合之前五次实验 (除 GPU 实验) 之外的所有并行化算法进行了对比分析, 在 x86 平台下进行了数据规模分别为 1024,2048,3072,4096, 进程数目为 4, 线程数目为 4, 各种算法的性能如下图所示:

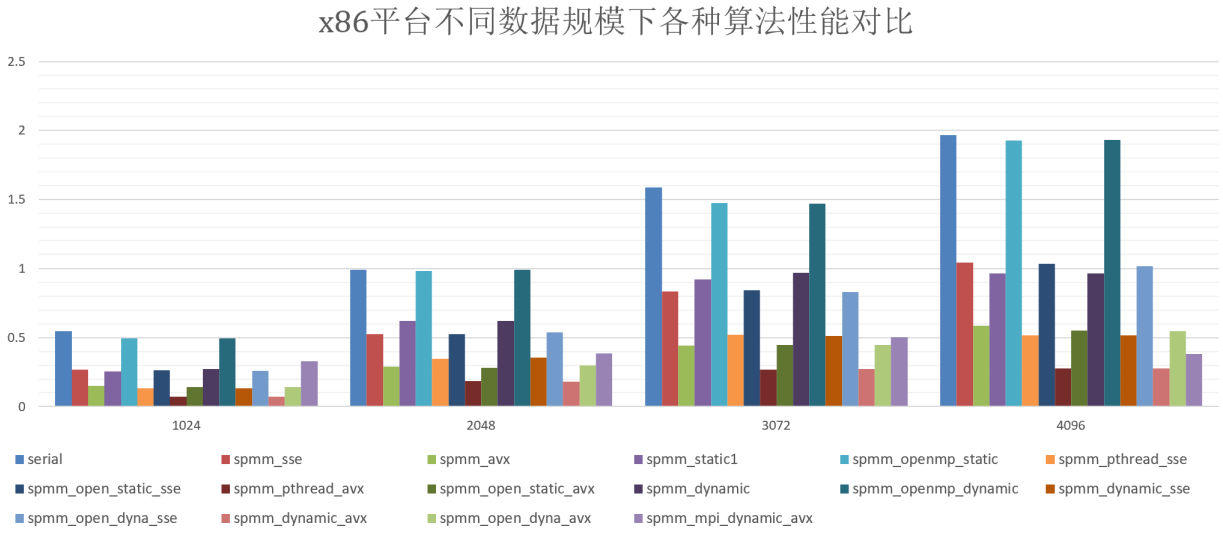


图 3.3: x86 平台上不同数据规模各种算法的程序性能对比

可以发现在这几种数据规模当中, 最优算法是静态 pthread 和 avx 结合以及动态 pthread 和 avx 结合的算法, 一部分原因可能是数据规模较小没有体现出 openMP 和 MPI 的威力, 另一部分原因可能是没有将 MPI 算法结合 pthread 进行并行化, 没有对 spMM 进行彻底的 MPI 算法并行化, 等待在大作业的工作中继续进一步完善算法, 进一步提升效率.

perf 工具和火焰图分析如下

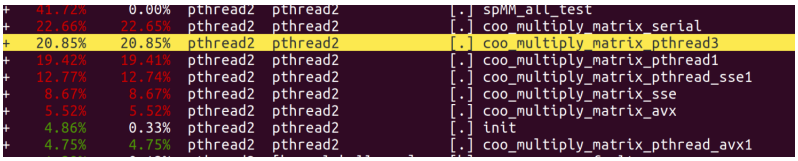


图 3.4: 在 x86 平台并行算法 perf 分析

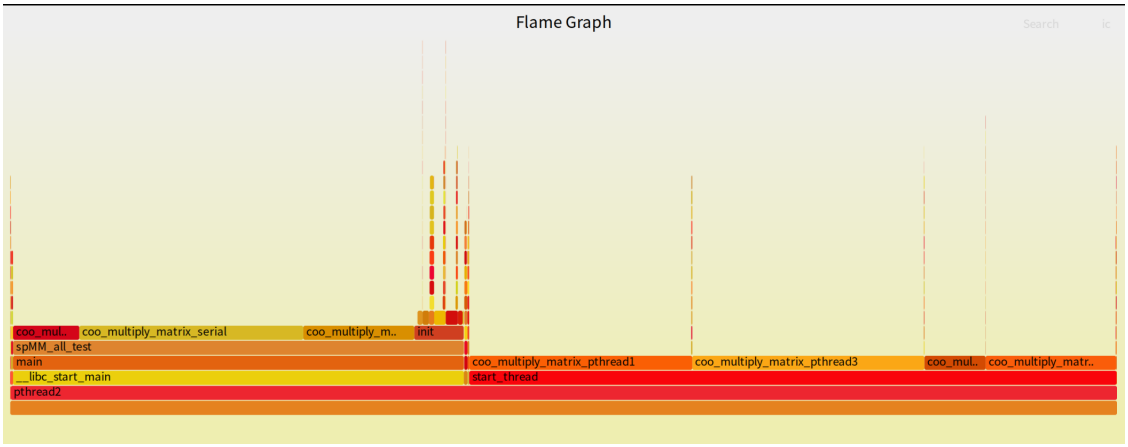


图 3.5: 在 x86 平台并行算法火焰图分析

4 代码上传

全部代码素材已上传GitHub.