MPI编程实验报告

> 学号：1913155　　姓名：袁懿

1.实验选题：LU分解

[默认选题]

2.实验目的：

①以LU分解作为切入点实践 MPI 编程；

②探究不同的**数据任务（行/列）划分**对于程序性能的影响；

③探究**使用** $MPI+Pthread$ **和** $MPI+OpenMP$ 给程序带来的影响；

④探究使用**不同的** $SIMD$（AVX/SSE） **策略**对程序的影响；

3.实验方案：

①学习 MPI 编程，复习 $OpenMP$ 与 $Pthread$ 与 $SIMD$ 的相关知识。

②使用同一台电脑同一个编译器进行编程，减小因为环境带来的误差。

③结合前面的几次实验 $SIMD$、$Pthread$、$OpenMP$ 的实验代码，在该基础上编写 $OpenMP$ 程序。

④根据③中编写的代码，设置相应的对比实验突出实验目的：

设置的对比实验如下：

(i) **探究使用** $MPI+omp/Pthread$ **对于原来的** MPI **程序的差异**

> - `normal_col.cpp`
> - `MPI_omp_col.cpp`
> - `MPI_pthread_col.cpp`

(ii) **对比** omp **和** $Ptrhead$ **两者结合** MPI **之后的两者区别**

> - `MPI_omp_SSE_row.cpp`
> - `MPI_pthread_SSE_row.cpp`

(iii) **对比使用在** omp **与** $Pthread$ **中使用不同的** $SIMD$ **策略带来的影响**

> - `MPI_omp_SSE_row.cpp`
> - `MPI_omp_AVX_row.cpp`
> - `MPT_pthread_SSE_row.cpp`
> - `MPI_pthread_AVX_row.cpp`

(iv) 探究任务的划分（行/列）对于 $MPI+omp/Pthread$程序的影响

> - `MPI_omp_SSE_col.cpp`
> - `MPI_omp_SSE_rwo.cpp`
> - `MPI_pthread_SSE_col.cpp`
> - `MPI_pthread_SSE_row.cpp`

⑤按照上面的对比实验的结果，相应的控制变量，对此进行实验取平均值以减小误差，将所有的实验结果都记录下来。

⑥根据④⑤中得到的实验结果数据，分析原因。

⑦总结实验。

4.实验中的相关数据和代码

以下涉及大量重复代码（其实可以跳过，可以直接看思路，因为代码的量实在是太大了）

4.1 对比实验一：

探究使用 $MPI+omp/Pthread$ 对于原来的 MPI 程序的差异

（控制线程数$n=4$，并改变数据规模的大小，对此实验取平均值）

> - `normal_col.cpp`
> - `MPI_omp_col.cpp`
> - `MPI_pthread_col.cpp`

code_1:

`normal_col.main`

```
int main(int argc,char* argv[]){
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    pthread_barrier_init(&childbarrier_row, NULL,thread_count+1);
    pthread_barrier_init(&childbarrier_col,NULL, thread_count+1);
    sem_init(&sem_parent, 0, 0);
    pthread_t threadID[thread_count];

    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    //int distributerow=n/(numprocs-1);
    int distributecol=n/(numprocs);
    //0号进程首先完成初始化的工作，再将按行划分的每一行传给不同的进程
    if(myid==0)
    {
        init();//初始化
        //自定义数据类型
        for(int i=1;i< numprocs;i++)
        {
            int begin=i*distributecol;
```

```
            int end=begin+distributecol;
            if(i==numprocs-1)
                end=n;
            MPI_Datatype block;
            MPI_Type_vector(n,(end-begin),n,MPI_FLOAT,&block);
            MPI_Type_commit(&block);
            MPI_Send((void *)(A[0]+begin),1,block,i,0,MPI_COMM_WORLD);
        }
        //printA();
    }
    else//接受消息后并更新对应的矩阵
    {
        int begin=myid*distributecol;
        int end=begin+distributecol;
        if(myid==numprocs-1)
            end=n;
        MPI_Datatype block;
        MPI_Type_vector(n,(end-begin),n,MPI_FLOAT,&block);
        MPI_Type_commit(&block);
        MPI_Recv((void *)
(A[0]+begin),1,block,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }

    //开始进行消去
    int begin=(myid)*distributecol;
    int end=begin+distributecol;
    if(myid==numprocs-1)
        end=n;
    for(int k=0;k<n;k++)
    {
        int source=k/distributecol;//注意source
        if(source>=numprocs)
            source=numprocs-1;
        MPI_Datatype temcol;
        MPI_Type_vector(n-k,1,n,MPI_FLOAT,&temcol);
        MPI_Type_commit(&temcol);
        MPI_Bcast((void *)(A[k]+k),1,temcol,source,MPI_COMM_WORLD);//将用于矩阵更新
的列传出

        for(int j=(begin>=(k+1)?begin:(k+1));j<end;j++)
            A[k][j]=A[k][j]/A[k][k];
        A[k][k]=1;

        for(int j=k+1;j<n;j++)
        {
            for(int i=(begin>=(k+1)?begin:(k+1));i<end;i++)
            {
                A[j][i]=A[j][i]-A[j][k]*A[k][i];
            }
            A[j][k]=0;
        }

    }


    if(myid!=0)
    {
        //将每个进程更新后的结果传回
```

```
        //MPI_Send((void *)A[begin],count,MPI_FLOAT,0,1,MPI_COMM_WORLD);
        MPI_Datatype block;
        MPI_Type_vector(n,(end-begin),n,MPI_FLOAT,&block);
        MPI_Type_commit(&block);
        MPI_Send((void *)(A[0]+begin),1,block,0,1,MPI_COMM_WORLD);
    }
    else
    {
        for(int i=1;i<numprocs;i++)
        {
            int begin=i*distributecol;
            int end=begin+distributecol;
            if(i==numprocs-1)
                end=n;
            MPI_Datatype block;
            MPI_Type_vector(n,(end-begin),n,MPI_FLOAT,&block);
            MPI_Type_commit(&block);
            MPI_Recv((void *)
(A[0]+begin),1,block,i,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
    }
    if(myid==0)
        printA();

    MPI_Finalize();

}
```

code_2:

`MPI_omp_col.main`

```
int main(int argc,char* argv[]){
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    pthread_barrier_init(&childbarrier_row, NULL,thread_count+1);
    pthread_barrier_init(&childbarrier_col,NULL, thread_count+1);
    sem_init(&sem_parent, 0, 0);
    pthread_t threadID[thread_count];

    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    //int distributerow=n/(numprocs-1);
    int distributecol=n/(numprocs);
    //0号进程首先完成初始化的工作，再将按行划分的每一行传给不同的进程
    if(myid==0)
    {
        init();//初始化
        //自定义数据类型
        for(int i=1;i< numprocs;i++)
        {
            int begin=i*distributecol;
            int end=begin+distributecol;
            if(i==numprocs-1)
                end=n;
```

```
            MPI_Datatype block;
            MPI_Type_vector(n,(end-begin),n,MPI_FLOAT,&block);
            MPI_Type_commit(&block);
            MPI_Send((void *)(A[0]+begin),1,block,i,0,MPI_COMM_WORLD);
        }
        //printA();
    }
    else//接受消息后并更新对应的矩阵
    {
        int begin=myid*distributecol;
        int end=begin+distributecol;
        if(myid==numprocs-1)
            end=n;
        MPI_Datatype block;
        MPI_Type_vector(n,(end-begin),n,MPI_FLOAT,&block);
        MPI_Type_commit(&block);
        MPI_Recv((void *)
(A[0]+begin),1,block,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }

    //开始进行消去
    int begin=(myid)*distributecol;
    int end=begin+distributecol;
    if(myid==numprocs-1)
        end=n;
    # pragma omp parallel for num_threads(thread_count)\
    shared(A)
    for(int k=0;k<n;k++)
    {
        int source=k/distributecol;//注意source
        if(source>=numprocs)
            source=numprocs-1;
        MPI_Datatype temcol;
        MPI_Type_vector(n-k,1,n,MPI_FLOAT,&temcol);
        MPI_Type_commit(&temcol);
        MPI_Bcast((void *)(A[k]+k),1,temcol,source,MPI_COMM_WORLD);//将用于矩阵更新
的列传出

        for(int j=(begin>=(k+1)?begin:(k+1));j<end;j++)
            A[k][j]=A[k][j]/A[k][k];
        A[k][k]=1;

        for(int j=k+1;j<n;j++)
        {
            for(int i=(begin>=(k+1)?begin:(k+1));i<end;i++)
            {
                A[j][i]=A[j][i]-A[j][k]*A[k][i];
            }
            A[j][k]=0;
        }

    }


    if(myid!=0)
    {
        //将每个进程更新后的结果传回
        //MPI_Send((void *)A[begin],count,MPI_FLOAT,0,1,MPI_COMM_WORLD);
```

```
        MPI_Datatype block;
        MPI_Type_vector(n,(end-begin),n,MPI_FLOAT,&block);
        MPI_Type_commit(&block);
        MPI_Send((void *)(A[0]+begin),1,block,0,1,MPI_COMM_WORLD);
    }
    else
    {
        for(int i=1;i<numprocs;i++)
        {
            int begin=i*distributecol;
            int end=begin+distributecol;
            if(i==numprocs-1)
                end=n;
            MPI_Datatype block;
            MPI_Type_vector(n,(end-begin),n,MPI_FLOAT,&block);
            MPI_Type_commit(&block);
            MPI_Recv((void *)
(A[0]+begin),1,block,i,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
    }
    if(myid==0)
        printA();

    MPI_Finalize();

}
```

code_3:

`MPI_ptrhead_col.main`

```
int main(int argc,char* argv[]){
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    pthread_barrier_init(&childbarrier_row, NULL,thread_count+1);
    pthread_barrier_init(&childbarrier_col,NULL, thread_count+1);
    sem_init(&sem_parent, 0, 0);
    pthread_t threadID[thread_count];

    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    //int distributerow=n/(numprocs-1);
    int distributecol=n/(numprocs);
    //0号进程首先完成初始化的工作，再将按行划分的每一行传给不同的进程
    if(myid==0)
    {
        init();//初始化
        //自定义数据类型
        for(int i=1;i< numprocs;i++)
        {
            int begin=i*distributecol;
            int end=begin+distributecol;
            if(i==numprocs-1)
                end=n;
            MPI_Datatype block;
```

```
            MPI_Type_vector(n,(end-begin),n,MPI_FLOAT,&block);
            MPI_Type_commit(&block);
            MPI_Send((void *)(A[0]+begin),1,block,i,0,MPI_COMM_WORLD);
        }
        //printA();
    }
    else//接受消息后并更新对应的矩阵
    {
        int begin=myid*distributecol;
        int end=begin+distributecol;
        if(myid==numprocs-1)
            end=n;
        MPI_Datatype block;
        MPI_Type_vector(n,(end-begin),n,MPI_FLOAT,&block);
        MPI_Type_commit(&block);
        MPI_Recv((void *)
(A[0]+begin),1,block,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }

    //开始进行消去
    int begin=(myid)*distributecol;
    int end=begin+distributecol;
    if(myid==numprocs-1)
        end=n;
    for(int k=0;k<n;k++)
    {
        int source=k/distributecol;//注意source
        if(source>=numprocs)
            source=numprocs-1;
        MPI_Datatype temcol;
        MPI_Type_vector(n-k,1,n,MPI_FLOAT,&temcol);
        MPI_Type_commit(&temcol);
        // if(k==1&&myid==1)
        //     printA();
        MPI_Bcast((void *)(A[k]+k),1,temcol,source,MPI_COMM_WORLD);//将用于矩阵更新
的列传出
        // if(k==0&&myid==1)
        //     printA();
        if(k==0)
        {
            for(int i=0;i<thread_count;i++)
            {
                datagroups[i].id=i;
                datagroups[i].begin=begin;
                datagroups[i].end=end;
                datagroups[i].myid=myid;
                pthread_create(&threadID[i],NULL,dealwithbycol_SSE,
(void*)&datagroups[i]);
            }
        }
        else
        {
            pthread_barrier_wait(&childbarrier_col);
        }
        for(int i=0;i<thread_count;i++)
        {
            sem_wait(&sem_parent);
        }
```

```
            A[k][k]=1;
            for(int i=k+1;i<n;i++)
                A[i][k]=0;
        }
        pthread_barrier_wait(&childbarrier_col);
        for(int i=0;i<thread_count;i++)
        {
            pthread_join(threadID[i],NULL);
        }


        if(myid!=0)
        {
            //将每个进程更新后的结果传回
            //MPI_Send((void *)A[begin],count,MPI_FLOAT,0,1,MPI_COMM_WORLD);
            MPI_Datatype block;
            MPI_Type_vector(n,(end-begin),n,MPI_FLOAT,&block);
            MPI_Type_commit(&block);
            MPI_Send((void *)(A[0]+begin),1,block,0,1,MPI_COMM_WORLD);
        }
        else
        {
            for(int i=1;i<numprocs;i++)
            {
                int begin=i*distributecol;
                int end=begin+distributecol;
                if(i==numprocs-1)
                    end=n;
                MPI_Datatype block;
                MPI_Type_vector(n,(end-begin),n,MPI_FLOAT,&block);
                MPI_Type_commit(&block);
                MPI_Recv((void *)
(A[0]+begin),1,block,i,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
            }
        }
        if(myid==0)
            printA();

        MPI_Finalize();

}
```

4.2 对比实验二:

探究使用 $MPI+omp$ 和 $MPI+Pthread$ 两者之间的不同

（控制线程数相等，改变不同的数据规模，进行多次实验取平均值）

> - `MPI_omp_SSE_row.cpp`
> - `MPI_pthread_SSE_row.cpp`

code_1:

`MPI_omp_SSE_row.main`

```
int main(int argc,char* argv[]){
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    pthread_barrier_init(&childbarrier_row, NULL,thread_count+1);
    pthread_barrier_init(&childbarrier_col,NULL, thread_count+1);
    sem_init(&sem_parent, 0, 0);
    pthread_t threadID[thread_count];

    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    int distributerow=n/(numprocs-1);
    //0号进程首先完成初始化的工作，再将按行划分的每一行传给不同的进程
    if(myid==0)
    {
        init();//初始化
        //任务分发
        for(int i=1;i<numprocs;i++)//从1号开始分发
        {
            int begin=(i-1)*distributerow;
            int end=begin+distributerow;
            if(i==numprocs-1)
                end=n;
            int count=(end-begin)*n;//发送数据个数
            //从begin行开始传
            MPI_Send((void *)A[begin],count,MPI_FLOAT,i,0,MPI_COMM_WORLD);
        }
        printA();
    }
    else//接受消息后并更新对应的矩阵
    {
        int begin=(myid-1)*distributerow;
        int end=begin+distributerow;
        if(myid==numprocs-1)
            end=n;
        int count=(end-begin)*n;
        MPI_Recv((void
*)A[begin],count,MPI_FLOAT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }

    //开始进行消去
    int begin=(myid-1)*distributerow;
    int end=begin+distributerow;
    if(myid==numprocs-1)
        end=n;
    int count=(end-begin)*n;


    for(int k=0;k<n;k++)
    {
        if(myid==0)
        {
            if(k!=0)
            {
                int source=(k/distributerow+1)<(numprocs-1)?(k/distributerow+1):
(numprocs-1);
```

```
                MPI_Recv((void *)(A[k]+k), n-k, MPI_FLOAT, source, 2,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
            __m128 t1,t2,t3;
            int preprocessnumber=(n-k-1)%4;
            int begincol=k+1+preprocessnumber;
            float head[4]={A[k][k],A[k][k],A[k][k],A[k][k]};
            t2=_mm_loadu_ps(head);
            for(int j=k+1;j<k+1+preprocessnumber;j++)
            {
                A[k][j]=A[k][j]/A[k][k];
            }
            # pragma omp parallel for num_threads(thread_count)\
            shared(A)
            for(int j=begincol;j<n;j=j+4)
            {
                t1=_mm_loadu_ps(A[k]+j);
                t1=_mm_div_ps(t1,t2);
                _mm_storeu_ps(A[k]+j,t1);
            }
            A[k][k]=1;
        }
        //数据广播，不能放到if语句，采用规约树广播，此时共享
        MPI_Bcast((void *)(A[k]+k),n-k,MPI_FLOAT,0,MPI_COMM_WORLD);

        //收到数据后才能跑
        if(myid!=0)
        {

            // for(int j=(begin>(k+1)?begin:k+1);j<end;j++)//注意begin与i+1的关系
            // {
            //     for(int i=k+1;i<n;i++)
            //     {
            //         A[j][i]=A[j][i]-A[j][k]*A[k][i];
            //     }
            //     A[j][k]=0;
            // }

            //将begin与end的数据均匀的分给线程的数量
            __m128 t1,t2,t3;
            int preprocessnumber=(n-k-1)%4;
            int begincol=k+1+preprocessnumber;
            for(int i=k+1;i<n;i++)
            {
                for(int j=k+1;j<k+1+preprocessnumber;j++)
                {
                    A[i][j]=A[i][j]-A[i][k]*A[k][j];
                }
                A[i][k]=0;
            }
            # pragma omp parallel for num_threads(thread_count)\
            shared(A)
            for(int i=k+1;i<n;i++)
            {
                float head1[4]={A[i][k],A[i][k],A[i][k],A[i][k]};
                t3=_mm_loadu_ps(head1);
                for(int j=begincol;j<n;j+=4)
                {
```

```
                    t1=_mm_loadu_ps(A[k]+j);
                    t2=_mm_loadu_ps(A[i]+j);
                    t1=_mm_mul_ps(t1,t3);
                    t2=_mm_sub_ps(t2,t1);
                    _mm_storeu_ps(A[i]+j,t2);
                }
                A[i][k]=0;
            }
            if((k+1<n)&&(k+1)>=begin&&(k+1)<end)//更新的数据传回0号
            {
                MPI_Send((void *)(A[k+1]+k+1), n-(k+1), MPI_FLOAT, 0, 2,
MPI_COMM_WORLD);
            }
        }
    }
    if(myid!=0)
    {
        //将每个进程更新后的结果传回
        MPI_Send((void *)A[begin],count,MPI_FLOAT,0,1,MPI_COMM_WORLD);
    }
    else
    {
        for(int i=1;i<numprocs;i++)
        {
            int begin=(i-1)*distributerow;
            int end=begin+distributerow;
            if(i==numprocs-1)
                end=n;
            int count=(end-begin)*n;//发送数据个数
            MPI_Recv((void
*)A[begin],count,MPI_FLOAT,i,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
    }
    if(myid==0)
            printA();
    MPI_Finalize();
}
```

code_2:

`MPI_pthread_SSE_row.dealwithbyrow_SSE`

```
void * dealwithbyrow_SSE(void * datai)
{
    data* datagroup= (data*)datai;
    __m128 t1,t2,t3;
    for(int k=0;k<n;k++)
    {

        int begin=datagroup->begin +datagroup->id*((datagroup->end-datagroup-
>begin)/thread_count);
        int end=begin+(datagroup->end-datagroup->begin)/thread_count;
        if(datagroup->id==thread_count-1)
            end=datagroup->end;
        int preprocessnumber=(n-k-1)%4;
        int begincol=k+1+preprocessnumber;
```

```
        for(int i=(begin>=(k+1)?begin:k+1);i<end;i++)
        {
            for(int j=k+1;j<n;j++)
            {
                    A[i][j]=A[i][j]-A[i][k]*A[k][j];
            }

            A[i][k]=0;
        }

        for(int i=(begin>=(k+1)?begin:(k+1));i<end;i++)
        {
            float head1[4]={A[i][k],A[i][k],A[i][k],A[i][k]};
            t3=_mm_loadu_ps(head1);
            for(int j=begincol;j<n;j+=4)
            {
                t1=_mm_loadu_ps(A[k]+j);
                t2=_mm_loadu_ps(A[i]+j);
                t1=_mm_mul_ps(t1,t3);
                t2=_mm_sub_ps(t2,t1);
                _mm_storeu_ps(A[i]+j,t2);
            }
            A[i][k]=0;
        }
        sem_post(&sem_parent);
        pthread_barrier_wait(&childbarrier_row);
    }
    pthread_exit(NULL);
}
```

`MPI_pthread_SSE_row.main`

```
int main(int argc,char* argv[]){
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    pthread_barrier_init(&childbarrier_row, NULL,thread_count+1);
    pthread_barrier_init(&childbarrier_col,NULL, thread_count+1);
    sem_init(&sem_parent, 0, 0);
    pthread_t threadID[thread_count];

    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    int distributerow=n/(numprocs-1);
    //0号进程首先完成初始化的工作，再将按行划分的每一行传给不同的进程
    if(myid==0)
    {
        init();//初始化
        //任务分发
        for(int i=1;i<numprocs;i++)//从1号开始分发
        {
            int begin=(i-1)*distributerow;
            int end=begin+distributerow;
            if(i==numprocs-1)
                end=n;
            int count=(end-begin)*n;//发送数据个数
```

```
            //从begin行开始传
            MPI_Send((void *)A[begin],count,MPI_FLOAT,i,0,MPI_COMM_WORLD);
        }
        printA();
    }
    else//接受消息后并更新对应的矩阵
    {
        int begin=(myid-1)*distributerow;
        int end=begin+distributerow;
        if(myid==numprocs-1)
            end=n;
        int count=(end-begin)*n;
        MPI_Recv((void
*)A[begin],count,MPI_FLOAT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }

    //开始进行消去
    int begin=(myid-1)*distributerow;
    int end=begin+distributerow;
    if(myid==numprocs-1)
        end=n;
    int count=(end-begin)*n;
    for(int k=0;k<n;k++)
    {
        if(myid==0)
        {
            if(k!=0)
            {
                int source=(k/distributerow+1)<(numprocs-1)?(k/distributerow+1):
(numprocs-1);
                MPI_Recv((void *)(A[k]+k), n-k, MPI_FLOAT, source, 2,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
            __m128 t1,t2,t3;
            int preprocessnumber=(n-k-1)%4;
            int begincol=k+1+preprocessnumber;
            float head[4]={A[k][k],A[k][k],A[k][k],A[k][k]};
            t2=_mm_loadu_ps(head);
            for(int j=k+1;j<k+1+preprocessnumber;j++)
            {
                A[k][j]=A[k][j]/A[k][k];
            }
            for(int j=begincol;j<n;j=j+4)
            {
                t1=_mm_loadu_ps(A[k]+j);
                t1=_mm_div_ps(t1,t2);
                _mm_storeu_ps(A[k]+j,t1);
            }
            A[k][k]=1;


            // for(int j=i+1;j<n;j++)
            //     A[i][j]=A[i][j]/A[i][i];
            // //将更新完的行传给剩余的进程
            // A[i][i]=1;


        }
```

```
        //数据广播，不能放到if语句，采用规约树广播，此时共享
        MPI_Bcast((void *)(A[k]+k),n-k,MPI_FLOAT,0,MPI_COMM_WORLD);

        //收到数据后才能跑
        if(myid!=0)
        {
            // for(int j=(begin>(k+1)?begin:k+1);j<end;j++)//注意begin与i+1的关系
            // {
            //     for(int i=k+1;i<n;i++)
            //     {
            //         A[j][i]=A[j][i]-A[j][k]*A[k][i];
            //     }
            //     A[j][k]=0;
            // }

            //将begin与end的数据均匀的分给线程的数量
            if(k==0)
            {
                for(int i=0;i<thread_count;i++)
                {
                    datagroups[i].id=i;
                    datagroups[i].begin=begin>(k+1)?begin:k+1;
                    datagroups[i].end=end;
                    datagroups[i].myid=myid;
                    pthread_create(&threadID[i],NULL,dealwithbyrow_SSE,
(void*)&datagroups[i]);
                }
            }
            else
                pthread_barrier_wait(&childbarrier_row);
            for(int i=0;i<thread_count;i++)//
            {
                sem_wait(&sem_parent);
            }

            if((k+1<n)&&(k+1)>=begin&&(k+1)<end)//更新的数据传回0号
            {
                MPI_Send((void *)(A[k+1]+k+1), n-(k+1), MPI_FLOAT, 0, 2,
MPI_COMM_WORLD);
            }
        }
    }
    if(myid!=0)
    {
        pthread_barrier_wait(&childbarrier_row);
        for(int i=0;i<thread_count;i++)
        {
            pthread_join(threadID[i],NULL);
        }
        //将每个进程更新后的结果传回
        MPI_Send((void *)A[begin],count,MPI_FLOAT,0,1,MPI_COMM_WORLD);
    }
    else
    {
        for(int i=1;i<numprocs;i++)
        {
            int begin=(i-1)*distributerow;
            int end=begin+distributerow;
```

```
            if(i==numprocs-1)
                end=n;
            int count=(end-begin)*n;//发送数据个数
            MPI_Recv((void
*)A[begin],count,MPI_FLOAT,i,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
    }
    if(myid==0)
            printA();
    MPI_Finalize();

}
```

4.3 对比实验三：

对比使用在 omp 与 $Pthread$ 中使用不同的 $SIMD$ 策略带来的影响

> - `MPI_omp_SSE_row.cpp`
> - `MPI_omp_AVX_row.cpp`
> - `MPT_pthread_SSE_row.cpp`
> - `MPI_pthread_AVX_row.cpp`

code_1:

`MPI_omp_SSE_row.main`

```
int main(int argc,char* argv[]){
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    pthread_barrier_init(&childbarrier_row, NULL,thread_count+1);
    pthread_barrier_init(&childbarrier_col,NULL, thread_count+1);
    sem_init(&sem_parent, 0, 0);
    pthread_t threadID[thread_count];

    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    int distributerow=n/(numprocs-1);
    //0号进程首先完成初始化的工作，再将按行划分的每一行传给不同的进程
    if(myid==0)
    {
        init();//初始化
        //任务分发
        for(int i=1;i<numprocs;i++)//从1号开始分发
        {
            int begin=(i-1)*distributerow;
            int end=begin+distributerow;
            if(i==numprocs-1)
                end=n;
            int count=(end-begin)*n;//发送数据个数
            //从begin行开始传
            MPI_Send((void *)A[begin],count,MPI_FLOAT,i,0,MPI_COMM_WORLD);
        }
        printA();
```

```
    }
    else//接受消息后并更新对应的矩阵
    {
        int begin=(myid-1)*distributerow;
        int end=begin+distributerow;
        if(myid==numprocs-1)
            end=n;
        int count=(end-begin)*n;
        MPI_Recv((void
*)A[begin],count,MPI_FLOAT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }

    //开始进行消去
    int begin=(myid-1)*distributerow;
    int end=begin+distributerow;
    if(myid==numprocs-1)
        end=n;
    int count=(end-begin)*n;


    for(int k=0;k<n;k++)
    {
        if(myid==0)
        {
            if(k!=0)
            {
                int source=(k/distributerow+1)<(numprocs-1)?(k/distributerow+1):
(numprocs-1);
                MPI_Recv((void *)(A[k]+k), n-k, MPI_FLOAT, source, 2,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
            __m128 t1,t2,t3;
            int preprocessnumber=(n-k-1)%4;
            int begincol=k+1+preprocessnumber;
            float head[4]={A[k][k],A[k][k],A[k][k],A[k][k]};
            t2=_mm_loadu_ps(head);
            for(int j=k+1;j<k+1+preprocessnumber;j++)
            {
                A[k][j]=A[k][j]/A[k][k];
            }
            # pragma omp parallel for num_threads(thread_count)\
            shared(A)
            for(int j=begincol;j<n;j=j+4)
            {
                t1=_mm_loadu_ps(A[k]+j);
                t1=_mm_div_ps(t1,t2);
                _mm_storeu_ps(A[k]+j,t1);
            }
            A[k][k]=1;
        }
        //数据广播，不能放到if语句，采用规约树广播，此时共享
        MPI_Bcast((void *)(A[k]+k),n-k,MPI_FLOAT,0,MPI_COMM_WORLD);

        //收到数据后才能跑
        if(myid!=0)
        {

            // for(int j=(begin>(k+1)?begin:k+1);j<end;j++)//注意begin与i+1的关系
```

```
            // {
            //     for(int i=k+1;i<n;i++)
            //     {
            //         A[j][i]=A[j][i]-A[j][k]*A[k][i];
            //     }
            //     A[j][k]=0;
            // }

            //将begin与end的数据均匀的分给线程的数量
            __m128 t1,t2,t3;
            int preprocessnumber=(n-k-1)%4;
            int begincol=k+1+preprocessnumber;
            for(int i=k+1;i<n;i++)
            {
                for(int j=k+1;j<k+1+preprocessnumber;j++)
                {
                    A[i][j]=A[i][j]-A[i][k]*A[k][j];
                }
                A[i][k]=0;
            }
            # pragma omp parallel for num_threads(thread_count)\
            shared(A)
            for(int i=k+1;i<n;i++)
            {
                float head1[4]={A[i][k],A[i][k],A[i][k],A[i][k]};
                t3=_mm_loadu_ps(head1);
                for(int j=begincol;j<n;j+=4)
                {
                    t1=_mm_loadu_ps(A[k]+j);
                    t2=_mm_loadu_ps(A[i]+j);
                    t1=_mm_mul_ps(t1,t3);
                    t2=_mm_sub_ps(t2,t1);
                    _mm_storeu_ps(A[i]+j,t2);
                }
                A[i][k]=0;
            }
            if((k+1<n)&&(k+1)>=begin&&(k+1)<end)//更新的数据传回0号
            {
                MPI_Send((void *)(A[k+1]+k+1), n-(k+1), MPI_FLOAT, 0, 2,
MPI_COMM_WORLD);
            }
        }
    }
    if(myid!=0)
    {
        //将每个进程更新后的结果传回
        MPI_Send((void *)A[begin],count,MPI_FLOAT,0,1,MPI_COMM_WORLD);
    }
    else
    {
        for(int i=1;i<numprocs;i++)
        {
            int begin=(i-1)*distributerow;
            int end=begin+distributerow;
            if(i==numprocs-1)
                end=n;
            int count=(end-begin)*n;//发送数据个数
```

```
            MPI_Recv((void
*)A[begin],count,MPI_FLOAT,i,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
    }
    if(myid==0)
            printA();
    MPI_Finalize();
}
```

code_2:

`MPI_omp_AVX_row.main`

```
int main(int argc,char* argv[]){
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    pthread_barrier_init(&childbarrier_row, NULL,thread_count+1);
    pthread_barrier_init(&childbarrier_col,NULL, thread_count+1);
    sem_init(&sem_parent, 0, 0);
    pthread_t threadID[thread_count];

    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    int distributerow=n/(numprocs-1);
    //0号进程首先完成初始化的工作，再将按行划分的每一行传给不同的进程
    if(myid==0)
    {
        init();//初始化
        //任务分发
        for(int i=1;i<numprocs;i++)//从1号开始分发
        {
            int begin=(i-1)*distributerow;
            int end=begin+distributerow;
            if(i==numprocs-1)
                end=n;
            int count=(end-begin)*n;//发送数据个数
            //从begin行开始传
            MPI_Send((void *)A[begin],count,MPI_FLOAT,i,0,MPI_COMM_WORLD);
        }
        printA();
    }
    else//接受消息后并更新对应的矩阵
    {
        int begin=(myid-1)*distributerow;
        int end=begin+distributerow;
        if(myid==numprocs-1)
            end=n;
        int count=(end-begin)*n;
        MPI_Recv((void
*)A[begin],count,MPI_FLOAT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }

    //开始进行消去
    int begin=(myid-1)*distributerow;
    int end=begin+distributerow;
```

```
    if(myid==numprocs-1)
        end=n;
    int count=(end-begin)*n;


    for(int k=0;k<n;k++)
    {
        if(myid==0)
        {
            if(k!=0)
            {
                int source=(k/distributerow+1)<(numprocs-1)?(k/distributerow+1):
(numprocs-1);
                MPI_Recv((void *)(A[k]+k), n-k, MPI_FLOAT, source, 2,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
            __m256 t1,t2,t3;
            int preprocessnumber=(n-k-1)%8;
            int begincol=k+1+preprocessnumber;
            float head[8]={A[k][k],A[k][k],A[k][k],A[k][k],A[k][k],A[k][k],A[k]
[k],A[k][k]};
            t2=_mm256_loadu_ps(head);
            for(int j=k+1;j<k+1+preprocessnumber;j++)
            {
                A[k][j]=A[k][j]/A[k][k];
            }
            # pragma omp parallel for num_threads(thread_count)\
            shared(A)
            for(int j=begincol;j<n;j=j+8)
            {
                t1=_mm256_loadu_ps(A[k]+j);
                t1=_mm256_div_ps(t1,t2);
                _mm256_storeu_ps(A[k]+j,t1);
            }
            A[k][k]=1;
        }
        //数据广播，不能放到if语句，采用规约树广播，此时共享
        MPI_Bcast((void *)(A[k]+k),n-k,MPI_FLOAT,0,MPI_COMM_WORLD);

        //收到数据后才能跑
        if(myid!=0)
        {

            // for(int j=(begin>(k+1)?begin:k+1);j<end;j++)//注意begin与i+1的关系
            // {
            //     for(int i=k+1;i<n;i++)
            //     {
            //         A[j][i]=A[j][i]-A[j][k]*A[k][i];
            //     }
            //     A[j][k]=0;
            // }

            //将begin与end的数据均匀的分给线程的数量
            __m256 t1,t2,t3;
            int preprocessnumber=(n-k-1)%8;
            int begincol=k+1+preprocessnumber;
            for(int i=k+1;i<n;i++)
            {
```

```
                for(int j=k+1;j<k+1+preprocessnumber;j++)
                {
                    A[i][j]=A[i][j]-A[i][k]*A[k][j];
                }
                A[i][k]=0;
            }
            # pragma omp parallel for num_threads(thread_count)\
            shared(A)
            for(int i=k+1;i<n;i++)
            {
                float head1[8]={A[i][k],A[i][k],A[i][k],A[i][k],A[i][k],A[i]
[k],A[i][k],A[i][k]};
                t3=_mm256_loadu_ps(head1);
                for(int j=begincol;j<n;j+=8)
                {
                    t1=_mm256_loadu_ps(A[k]+j);
                    t2=_mm256_loadu_ps(A[i]+j);
                    t1=_mm256_mul_ps(t1,t3);
                    t2=_mm256_sub_ps(t2,t1);
                    _mm256_storeu_ps(A[i]+j,t2);
                }
                A[i][k]=0;
            }
            if((k+1<n)&&(k+1)>=begin&&(k+1)<end)//更新的数据传回0号
            {
                MPI_Send((void *)(A[k+1]+k+1), n-(k+1), MPI_FLOAT, 0, 2,
MPI_COMM_WORLD);
            }
        }
    }
    if(myid!=0)
    {
        //将每个进程更新后的结果传回
        MPI_Send((void *)A[begin],count,MPI_FLOAT,0,1,MPI_COMM_WORLD);
    }
    else
    {
        for(int i=1;i<numprocs;i++)
        {
            int begin=(i-1)*distributerow;
            int end=begin+distributerow;
            if(i==numprocs-1)
                end=n;
            int count=(end-begin)*n;//发送数据个数
            MPI_Recv((void
*)A[begin],count,MPI_FLOAT,i,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
    }
    if(myid==0)
            printA();
    MPI_Finalize();
}
```

code_3:

`MPI_pthread_SSE_row.dealwithbyrow_SSE`

```
void * dealwithbyrow_SSE(void * datai)
{
    data* datagroup= (data*)datai;
    __m128 t1,t2,t3;
    for(int k=0;k<n;k++)
    {

        int begin=datagroup->begin +datagroup->id*((datagroup->end-datagroup-
>begin)/thread_count);
        int end=begin+(datagroup->end-datagroup->begin)/thread_count;
        if(datagroup->id==thread_count-1)
            end=datagroup->end;
        int preprocessnumber=(n-k-1)%4;
        int begincol=k+1+preprocessnumber;

        for(int i=(begin>=(k+1)?begin:k+1);i<end;i++)
        {
            for(int j=k+1;j<n;j++)
            {
                    A[i][j]=A[i][j]-A[i][k]*A[k][j];
            }

            A[i][k]=0;
        }

        for(int i=(begin>=(k+1)?begin:(k+1));i<end;i++)
        {
            float head1[4]={A[i][k],A[i][k],A[i][k],A[i][k]};
            t3=_mm_loadu_ps(head1);
            for(int j=begincol;j<n;j+=4)
            {
                t1=_mm_loadu_ps(A[k]+j);
                t2=_mm_loadu_ps(A[i]+j);
                t1=_mm_mul_ps(t1,t3);
                t2=_mm_sub_ps(t2,t1);
                _mm_storeu_ps(A[i]+j,t2);
            }
            A[i][k]=0;
        }
        sem_post(&sem_parent);
        pthread_barrier_wait(&childbarrier_row);
    }
    pthread_exit(NULL);
}
```

`MPI_pthread_SSE_row.main`

```
int main(int argc,char* argv[]){
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    pthread_barrier_init(&childbarrier_row, NULL,thread_count+1);
    pthread_barrier_init(&childbarrier_col,NULL, thread_count+1);
    sem_init(&sem_parent, 0, 0);
    pthread_t threadID[thread_count];

    int myid, numprocs;
    MPI_Init(&argc,&argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    int distributerow=n/(numprocs-1);
    //0号进程首先完成初始化的工作，再将按行划分的每一行传给不同的进程
    if(myid==0)
    {
        init();//初始化
        //任务分发
        for(int i=1;i<numprocs;i++)//从1号开始分发
        {
            int begin=(i-1)*distributerow;
            int end=begin+distributerow;
            if(i==numprocs-1)
                end=n;
            int count=(end-begin)*n;//发送数据个数
            //从begin行开始传
            MPI_Send((void *)A[begin],count,MPI_FLOAT,i,0,MPI_COMM_WORLD);
        }
        printA();
    }
    else//接受消息后并更新对应的矩阵
    {
        int begin=(myid-1)*distributerow;
        int end=begin+distributerow;
        if(myid==numprocs-1)
            end=n;
        int count=(end-begin)*n;
        MPI_Recv((void
*)A[begin],count,MPI_FLOAT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }

    //开始进行消去
    int begin=(myid-1)*distributerow;
    int end=begin+distributerow;
    if(myid==numprocs-1)
        end=n;
    int count=(end-begin)*n;
    for(int k=0;k<n;k++)
    {
        if(myid==0)
        {
            if(k!=0)
            {
                int source=(k/distributerow+1)<(numprocs-1)?(k/distributerow+1):
(numprocs-1);
                MPI_Recv((void *)(A[k]+k), n-k, MPI_FLOAT, source, 2,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
            __m128 t1,t2,t3;
            int preprocessnumber=(n-k-1)%4;
            int begincol=k+1+preprocessnumber;
            float head[4]={A[k][k],A[k][k],A[k][k],A[k][k]};
            t2=_mm_loadu_ps(head);
            for(int j=k+1;j<k+1+preprocessnumber;j++)
            {
                A[k][j]=A[k][j]/A[k][k];
            }
            for(int j=begincol;j<n;j=j+4)
```

```
            {
                t1=_mm_loadu_ps(A[k]+j);
                t1=_mm_div_ps(t1,t2);
                _mm_storeu_ps(A[k]+j,t1);
            }
            A[k][k]=1;


            // for(int j=i+1;j<n;j++)
            //     A[i][j]=A[i][j]/A[i][i];
            // //将更新完的行传给剩余的进程
            // A[i][i]=1;


        }
        //数据广播，不能放到if语句，采用规约树广播，此时共享
        MPI_Bcast((void *)(A[k]+k),n-k,MPI_FLOAT,0,MPI_COMM_WORLD);

        //收到数据后才能跑
        if(myid!=0)
        {
            // for(int j=(begin>(k+1)?begin:k+1);j<end;j++)//注意begin与i+1的关系
            // {
            //     for(int i=k+1;i<n;i++)
            //     {
            //         A[j][i]=A[j][i]-A[j][k]*A[k][i];
            //     }
            //     A[j][k]=0;
            // }

            //将begin与end的数据均匀的分给线程的数量
            if(k==0)
            {
                for(int i=0;i<thread_count;i++)
                {
                    datagroups[i].id=i;
                    datagroups[i].begin=begin>(k+1)?begin:k+1;
                    datagroups[i].end=end;
                    datagroups[i].myid=myid;
                    pthread_create(&threadID[i],NULL,dealwithbyrow_SSE,
(void*)&datagroups[i]);
                }
            }
            else
                pthread_barrier_wait(&childbarrier_row);
            for(int i=0;i<thread_count;i++)//
            {
                sem_wait(&sem_parent);
            }

            if((k+1<n)&&(k+1)>=begin&&(k+1)<end)//更新的数据传回0号
            {
                MPI_Send((void *)(A[k+1]+k+1), n-(k+1), MPI_FLOAT, 0, 2,
MPI_COMM_WORLD);
            }
        }
    }
    if(myid!=0)
```

```
    {
        pthread_barrier_wait(&childbarrier_row);
        for(int i=0;i<thread_count;i++)
        {
            pthread_join(threadID[i],NULL);
        }
        //将每个进程更新后的结果传回
        MPI_Send((void *)A[begin],count,MPI_FLOAT,0,1,MPI_COMM_WORLD);
    }
    else
    {
        for(int i=1;i<numprocs;i++)
        {
            int begin=(i-1)*distributerow;
            int end=begin+distributerow;
            if(i==numprocs-1)
                end=n;
            int count=(end-begin)*n;//发送数据个数
            MPI_Recv((void
*)A[begin],count,MPI_FLOAT,i,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
    }
    if(myid==0)
            printA();
    MPI_Finalize();

}
```

code_4:

`MPI_pthread_AVX_row.dealwithbyrow_AVX`

```
void * dealwithbyrow_AVX(void * datai)
{
    data* datagroup= (data*)datai;
    __m256 t1,t2,t3;
    for(int k=0;k<n;k++)
    {
        int begin=datagroup->begin +datagroup->id*((datagroup->end-datagroup-
>begin)/thread_count);
        int end=begin+(datagroup->end-datagroup->begin)/thread_count;
        if(datagroup->id==thread_count-1)
            end=datagroup->end;
        int preprocessnumber=(n-k-1)%8;
        int begincol=k+1+preprocessnumber;

        for(int i=(begin>=(k+1)?begin:k+1);i<end;i++)
        {
            for(int j=k+1;j<begincol;j++)
            {
                A[i][j]=A[i][j]-A[i][k]*A[k][j];
            }
            A[i][k]=0;
        }

        for(int i=(begin>=(k+1)?begin:(k+1));i<end;i++)
```

```
        {
            float head1[8]={A[i][k],A[i][k],A[i][k],A[i][k],A[i][k],A[i][k],A[i]
[k],A[i][k]};
            t3=_mm256_loadu_ps(head1);
            for(int j=begincol;j<n;j+=8)
            {
                t1=_mm256_loadu_ps(A[k]+j);
                t2=_mm256_loadu_ps(A[i]+j);
                t1=_mm256_mul_ps(t1,t3);
                t2=_mm256_sub_ps(t2,t1);
                _mm256_storeu_ps(A[i]+j,t2);
            }
            A[i][k]=0;
        }
        sem_post(&sem_parent);
        pthread_barrier_wait(&childbarrier_row);
    }
    pthread_exit(NULL);
}
```

MPI_pthread_AVX_row.main

```
int main(int argc,char* argv[]){
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    pthread_barrier_init(&childbarrier_row, NULL,thread_count+1);
    pthread_barrier_init(&childbarrier_col,NULL, thread_count+1);
    sem_init(&sem_parent, 0, 0);
    pthread_t threadID[thread_count];

    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    int distributerow=n/(numprocs-1);
    //0号进程首先完成初始化的工作，再将按行划分的每一行传给不同的进程
    if(myid==0)
    {
        init();//初始化
        //任务分发
        for(int i=1;i<numprocs;i++)//从1号开始分发
        {
            int begin=(i-1)*distributerow;
            int end=begin+distributerow;
            if(i==numprocs-1)
                end=n;
            int count=(end-begin)*n;//发送数据个数
            //从begin行开始传
            MPI_Send((void *)A[begin],count,MPI_FLOAT,i,0,MPI_COMM_WORLD);
        }
        printA();
    }
    else//接受消息后并更新对应的矩阵
    {
        int begin=(myid-1)*distributerow;
        int end=begin+distributerow;
        if(myid==numprocs-1)
            end=n;
```

```
        int count=(end-begin)*n;
        MPI_Recv((void
*)A[begin],count,MPI_FLOAT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }

    //开始进行消去
    int begin=(myid-1)*distributerow;
    int end=begin+distributerow;
    if(myid==numprocs-1)
        end=n;
    int count=(end-begin)*n;
    for(int k=0;k<n;k++)
    {
        if(myid==0)
        {
            if(k!=0)
            {
                int source=(k/distributerow+1)<(numprocs-1)?(k/distributerow+1):
(numprocs-1);
                MPI_Recv((void *)(A[k]+k), n-k, MPI_FLOAT, source, 2,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
            __m256 t1,t2,t3;
            int preprocessnumber=(n-k-1)%8;
            int begincol=k+1+preprocessnumber;
            float head[8]={A[k][k],A[k][k],A[k][k],A[k][k],A[k][k],A[k][k],A[k]
[k],A[k][k]};
            t2=_mm256_loadu_ps(head);
            for(int j=k+1;j<k+1+preprocessnumber;j++)
            {
                A[k][j]=A[k][j]/A[k][k];
            }
            for(int j=begincol;j<n;j+=8)
            {
                //A[k][j]=A[k][j]/A[k][k];
                t1=_mm256_loadu_ps(A[k]+j);
                t1=_mm256_div_ps(t1,t2);
                _mm256_storeu_ps(A[k]+j,t1);
            }
            A[k][k]=1;
            // for(int j=i+1;j<n;j++)
            //     A[i][j]=A[i][j]/A[i][i];
            // //将更新完的行传给剩余的进程
            // A[i][i]=1;


        }
        //数据广播，不能放到if语句，采用规约树广播，此时共享
        MPI_Bcast((void *)(A[k]+k),n-k,MPI_FLOAT,0,MPI_COMM_WORLD);

        //收到数据后才能跑
        if(myid!=0)
        {
            // for(int j=(begin>(k+1)?begin:k+1);j<end;j++)//注意begin与i+1的关系
            // {
            //     for(int i=k+1;i<n;i++)
            //     {
            //         A[j][i]=A[j][i]-A[j][k]*A[k][i];
```

```
            //     }
            //     A[j][k]=0;
            // }

            //将begin与end的数据均匀的分给线程的数量

            if(k==0)
            {
                for(int i=0;i<thread_count;i++)
                {
                    datagroups[i].id=i;
                    datagroups[i].begin=begin>(k+1)?begin:k+1;
                    datagroups[i].end=end;
                    datagroups[i].myid=myid;
                    pthread_create(&threadID[i],NULL,dealwithbyrow_AVX,
(void*)&datagroups[i]);
                }
            }
            else
                pthread_barrier_wait(&childbarrier_row);
            for(int i=0;i<thread_count;i++)//
            {
                sem_wait(&sem_parent);
            }
            if((k+1<n)&&(k+1)>=begin&&(k+1)<end)//更新的数据传回0号
            {
                MPI_Send((void *)(A[k+1]+k+1), n-(k+1), MPI_FLOAT, 0, 2,
MPI_COMM_WORLD);
            }
        }
    }
    if(myid!=0)
    {
        pthread_barrier_wait(&childbarrier_row);
        for(int i=0;i<thread_count;i++)
        {
            pthread_join(threadID[i],NULL);
        }
        //将每个进程更新后的结果传回
        MPI_Send((void *)A[begin],count,MPI_FLOAT,0,1,MPI_COMM_WORLD);
    }
    else
    {
        for(int i=1;i<numprocs;i++)
        {
            int begin=(i-1)*distributerow;
            int end=begin+distributerow;
            if(i==numprocs-1)
                end=n;
            int count=(end-begin)*n;//发送数据个数
            MPI_Recv((void
*)A[begin],count,MPI_FLOAT,i,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
    }
    if(myid==0)
            printA();
    MPI_Finalize();
```

```
}
```

4.4 对比实验四：

探究任务的划分（行/列）对于 $MPI+omp/Pthread$程序的影响

由于有大部分代码与上面的实验重复，并没有给出所有的代码

（控制线程数，改变数据规模的大小，多次实验取平均值）

> - `MPI_omp_SSE_col.cpp`
> - `MPI_omp_SSE_rwo.cpp`
> - `MPI_pthread_SSE_col.cpp`
> - `MPI_pthread_SSE_row.cpp`

code_1:

`MPI_omp_SSE_col.mian`

```
int main(int argc,char* argv[]){
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    pthread_barrier_init(&childbarrier_row, NULL,thread_count+1);
    pthread_barrier_init(&childbarrier_col,NULL, thread_count+1);
    sem_init(&sem_parent, 0, 0);
    pthread_t threadID[thread_count];

    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    //int distributerow=n/(numprocs-1);
    int distributecol=n/(numprocs);
    //0号进程首先完成初始化的工作，再将按行划分的每一行传给不同的进程
    if(myid==0)
    {
        init();//初始化
        //自定义数据类型
        for(int i=1;i< numprocs;i++)
        {
            int begin=i*distributecol;
            int end=begin+distributecol;
            if(i==numprocs-1)
                end=n;
            MPI_Datatype block;
            MPI_Type_vector(n,(end-begin),n,MPI_FLOAT,&block);
            MPI_Type_commit(&block);
            MPI_Send((void *)(A[0]+begin),1,block,i,0,MPI_COMM_WORLD);
        }
        //printA();
    }
    else//接受消息后并更新对应的矩阵
    {
        int begin=myid*distributecol;
        int end=begin+distributecol;
```

```
        if(myid==numprocs-1)
            end=n;
        MPI_Datatype block;
        MPI_Type_vector(n,(end-begin),n,MPI_FLOAT,&block);
        MPI_Type_commit(&block);
        MPI_Recv((void *)
(A[0]+begin),1,block,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }

    //开始进行消去
    int begin=(myid)*distributecol;
    int end=begin+distributecol;
    if(myid==numprocs-1)
        end=n;
    # pragma omp parallel for num_threads(thread_count)\
    shared(A)
    for(int k=0;k<n;k++)
    {
        int source=k/distributecol;//注意source
        if(source>=numprocs)
            source=numprocs-1;
        MPI_Datatype temcol;
        MPI_Type_vector(n-k,1,n,MPI_FLOAT,&temcol);
        MPI_Type_commit(&temcol);
        MPI_Bcast((void *)(A[k]+k),1,temcol,source,MPI_COMM_WORLD);//将用于矩阵更新
的列传出

        for(int j=(begin>=(k+1)?begin:(k+1));j<end;j++)
            A[k][j]=A[k][j]/A[k][k];
        A[k][k]=1;

        for(int j=k+1;j<n;j++)
        {
            for(int i=(begin>=(k+1)?begin:(k+1));i<end;i++)
            {
                A[j][i]=A[j][i]-A[j][k]*A[k][i];
            }
            A[j][k]=0;
        }

    }


    if(myid!=0)
    {
        //将每个进程更新后的结果传回
        //MPI_Send((void *)A[begin],count,MPI_FLOAT,0,1,MPI_COMM_WORLD);
        MPI_Datatype block;
        MPI_Type_vector(n,(end-begin),n,MPI_FLOAT,&block);
        MPI_Type_commit(&block);
        MPI_Send((void *)(A[0]+begin),1,block,0,1,MPI_COMM_WORLD);
    }
    else
    {
        for(int i=1;i<numprocs;i++)
        {
            int begin=i*distributecol;
            int end=begin+distributecol;
```

```
            if(i==numprocs-1)
                end=n;
            MPI_Datatype block;
            MPI_Type_vector(n,(end-begin),n,MPI_FLOAT,&block);
            MPI_Type_commit(&block);
            MPI_Recv((void *)
(A[0]+begin),1,block,i,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
    }
    if(myid==0)
        printA();

    MPI_Finalize();

}
```

code_2:

`MPI_omp_SSE_row.main`

```
int main(int argc,char* argv[]){
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    pthread_barrier_init(&childbarrier_row, NULL,thread_count+1);
    pthread_barrier_init(&childbarrier_col,NULL, thread_count+1);
    sem_init(&sem_parent, 0, 0);
    pthread_t threadID[thread_count];

    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    int distributerow=n/(numprocs-1);
    //0号进程首先完成初始化的工作，再将按行划分的每一行传给不同的进程
    if(myid==0)
    {
        init();//初始化
        //任务分发
        for(int i=1;i<numprocs;i++)//从1号开始分发
        {
            int begin=(i-1)*distributerow;
            int end=begin+distributerow;
            if(i==numprocs-1)
                end=n;
            int count=(end-begin)*n;//发送数据个数
            //从begin行开始传
            MPI_Send((void *)A[begin],count,MPI_FLOAT,i,0,MPI_COMM_WORLD);
        }
        printA();
    }
    else//接受消息后并更新对应的矩阵
    {
        int begin=(myid-1)*distributerow;
        int end=begin+distributerow;
        if(myid==numprocs-1)
            end=n;
        int count=(end-begin)*n;
```

```
        MPI_Recv((void
*)A[begin],count,MPI_FLOAT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }

    //开始进行消去
    int begin=(myid-1)*distributerow;
    int end=begin+distributerow;
    if(myid==numprocs-1)
        end=n;
    int count=(end-begin)*n;


    for(int k=0;k<n;k++)
    {
        if(myid==0)
        {
            if(k!=0)
            {
                int source=(k/distributerow+1)<(numprocs-1)?(k/distributerow+1):
(numprocs-1);
                MPI_Recv((void *)(A[k]+k), n-k, MPI_FLOAT, source, 2,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
            __m128 t1,t2,t3;
            int preprocessnumber=(n-k-1)%4;
            int begincol=k+1+preprocessnumber;
            float head[4]={A[k][k],A[k][k],A[k][k],A[k][k]};
            t2=_mm_loadu_ps(head);
            for(int j=k+1;j<k+1+preprocessnumber;j++)
            {
                A[k][j]=A[k][j]/A[k][k];
            }
            # pragma omp parallel for num_threads(thread_count)\
            shared(A)
            for(int j=begincol;j<n;j=j+4)
            {
                t1=_mm_loadu_ps(A[k]+j);
                t1=_mm_div_ps(t1,t2);
                _mm_storeu_ps(A[k]+j,t1);
            }
            A[k][k]=1;
        }
        //数据广播，不能放到if语句，采用规约树广播，此时共享
        MPI_Bcast((void *)(A[k]+k),n-k,MPI_FLOAT,0,MPI_COMM_WORLD);

        //收到数据后才能跑
        if(myid!=0)
        {

            // for(int j=(begin>(k+1)?begin:k+1);j<end;j++)//注意begin与i+1的关系
            // {
            //     for(int i=k+1;i<n;i++)
            //     {
            //         A[j][i]=A[j][i]-A[j][k]*A[k][i];
            //     }
            //     A[j][k]=0;
            // }
```

```
            //将begin与end的数据均匀的分给线程的数量
            __m128 t1,t2,t3;
            int preprocessnumber=(n-k-1)%4;
            int begincol=k+1+preprocessnumber;
            for(int i=k+1;i<n;i++)
            {
                for(int j=k+1;j<k+1+preprocessnumber;j++)
                {
                    A[i][j]=A[i][j]-A[i][k]*A[k][j];
                }
                A[i][k]=0;
            }
            # pragma omp parallel for num_threads(thread_count)\
            shared(A)
            for(int i=k+1;i<n;i++)
            {
                float head1[4]={A[i][k],A[i][k],A[i][k],A[i][k]};
                t3=_mm_loadu_ps(head1);
                for(int j=begincol;j<n;j+=4)
                {
                    t1=_mm_loadu_ps(A[k]+j);
                    t2=_mm_loadu_ps(A[i]+j);
                    t1=_mm_mul_ps(t1,t3);
                    t2=_mm_sub_ps(t2,t1);
                    _mm_storeu_ps(A[i]+j,t2);
                }
                A[i][k]=0;
            }
            if((k+1<n)&&(k+1)>=begin&&(k+1)<end)//更新的数据传回0号
            {
                MPI_Send((void *)(A[k+1]+k+1), n-(k+1), MPI_FLOAT, 0, 2,
MPI_COMM_WORLD);
            }
        }
    }
    if(myid!=0)
    {
        //将每个进程更新后的结果传回
        MPI_Send((void *)A[begin],count,MPI_FLOAT,0,1,MPI_COMM_WORLD);
    }
    else
    {
        for(int i=1;i<numprocs;i++)
        {
            int begin=(i-1)*distributerow;
            int end=begin+distributerow;
            if(i==numprocs-1)
                end=n;
            int count=(end-begin)*n;//发送数据个数
            MPI_Recv((void
*)A[begin],count,MPI_FLOAT,i,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
    }
    if(myid==0)
            printA();
    MPI_Finalize();
}
```

code_3:

`MPI_pthread_SSE_col.dealwithbycol_SSE`

```
void * dealwithbycol_SSE(void * datai)
{
    data* datagroup= (data*)datai;
    __m128 t1,t2,t3;

    for(int k=0;k<n;k++)
    {
        int begin=datagroup->begin +datagroup->id*((datagroup->end-datagroup-
>begin)/thread_count);
        int end=begin+(datagroup->end-datagroup->begin)/thread_count;
        if(datagroup->id==thread_count-1)
            end=datagroup->end;
        int preprocessnumber=(n-k-1)%4;
        int begincol=k+1+preprocessnumber;

        // if(k==0&&datagroup->myid==1)
        //     printA();
        for(int j=begin>=(k+1)?begin:(k+1);j<end;j++)
        {
            A[k][j]=A[k][j]/A[k][k];
        }
        // if(k==0&&datagroup->myid==1)
        //     printA();
        for(int j=k+1;j<n;j++)
        {
            for(int i=(begin>=(k+1)?begin:(k+1));i<end;i++)
            {
                A[j][i]=A[j][i]-A[j][k]*A[k][i];
            }
        }
        // if(k==1&&datagroup->myid==1)
        //     printA();
        sem_post(&sem_parent);
        pthread_barrier_wait(&childbarrier_col);

    }
    pthread_exit(NULL);
}
```

code_4:

`MPI_pthread_SSE_row.dealwithrow_SSE`

```
void * dealwithbyrow_SSE(void * datai)
{
    data* datagroup= (data*)datai;
    __m128 t1,t2,t3;
    for(int k=0;k<n;k++)
    {
```

```
        int begin=datagroup->begin +datagroup->id*((datagroup->end-datagroup-
>begin)/thread_count);
        int end=begin+(datagroup->end-datagroup->begin)/thread_count;
        if(datagroup->id==thread_count-1)
            end=datagroup->end;
        int preprocessnumber=(n-k-1)%4;
        int begincol=k+1+preprocessnumber;

        for(int i=(begin>=(k+1)?begin:k+1);i<end;i++)
        {
            for(int j=k+1;j<n;j++)
            {
                    A[i][j]=A[i][j]-A[i][k]*A[k][j];
            }

            A[i][k]=0;
        }

        for(int i=(begin>=(k+1)?begin:(k+1));i<end;i++)
        {
            float head1[4]={A[i][k],A[i][k],A[i][k],A[i][k]};
            t3=_mm_loadu_ps(head1);
            for(int j=begincol;j<n;j+=4)
            {
                t1=_mm_loadu_ps(A[k]+j);
                t2=_mm_loadu_ps(A[i]+j);
                t1=_mm_mul_ps(t1,t3);
                t2=_mm_sub_ps(t2,t1);
                _mm_store_ss(A[i]+j,t2);
            }
            A[i][k]=0;
        }
        sem_post(&sem_parent);
        pthread_barrier_wait(&childbarrier_row);
    }
    pthread_exit(NULL);
}
```

5.实验结论与结果分析：

5.1 对比实验一：

目的： 探究使用 $MPI+omp/Pthread$ 对于原来的 MPI 程序的差异

（相同的线程数相同的任务划分策略，不同的数据规模，多次实验后的结果记录）

实验1：最原始的 MPI

实验2：$MPI+omp$

实验3：$MPI+Pthread$

①实验结果：

数据规模	512	1024	2048
MPI	$108.234ms$	$826.237ms$	$7313.74ms$
$MPI+omp$	$32.5848ms$	$214.765ms$	$1907.31ms$
$MPI+Pthread$	$54.6312ms$	$232.234ms$	$1931.45ms$

(i) 当数据规模逐渐增大时，每个程序运行的时间不断增大，且是按照立方倍增大。

(ii) 使用 MPI 能够加快程序的运行速度。

(iii) 使用 $omp/Pthread$ 能加快原本的 MPI 程序的速度（大概四倍 $n=4$）。

(iv) 使用 $MPI+omp/Pthread$ 的方法能够大幅度加快程序运行的速度。

②结果分析：

(i) 数据规模不断增大，需要运算的数据与次数都会增加，由于原本的算法是一个 $\Theta(n^3)$ 的算法， 所以时间按照立方的速度增加。

(ii) 由于程序预设的线程数 $n=4$ 所以能够将程序的运行速度缩短为原来的四倍。

5.2 对比实验二：

目的： 对比 omp 和 $Ptrhead$ 两者结合 MPI 之后的两者区别

（控制线程数、数据规模、$SIMD$策略相同，改变数据规模，多次实验后取平均值记录结果）

实验1：$MPI+omp+SSE+row$

实验2：$MPI+Pthread+SSE+row$

①实验结果：

数据规模	512	1024	2048
$MPI+omp+SSE+row$	$10.2467ms$	$56.347ms$	$491.238ms$
$MPI+Pthread+SSE+row$	$14.8326ms$	$62.520ms$	$496.729ms$

(i) $MPI+omp$ 相较于 $MPI+Pthread$ 来说，程序运行的时间短一点。

(ii) 在数据规模比较小时，两者的差距比较大。

②结果分析：

(i) 使用的 omp 是通过编译器自动生成新的线程，而使用 $Pthread$ 是通过程序员手动生成新的线程，在生成线程的时间开销上 $Pthread$ 比 $OpenMP$ 要小

(ii) 由于两者调度的开销区别并不是很大，且两者的区别并不会随着数据规模的变化而有很大不同，当数据规模较小时，程序运行的时间本身就比较小，所以会导致看起来两者差异较大。

5.3 对比实验三：

目的： 对比使用在 omp 与 $Pthread$ 中使用不同的 $SIMD$ 策略带来的影响

（控制线程数、任务划分相同，改变数据规模的大小，记录实验的结果）

实验1：$MPI+omp+SSE+row$

实验2：$MPI+omp+AVX+row$

实验3：$MPI+Pthread+SSE+row$

实验4：$MPI+Pthread+AVX+row$

①实验结果：

数据规模	512	1024	2048
$MPI+omp+SSE+row$	$10.2467ms$	$56.347ms$	$491.238ms$
$MPI+omp+AVX+row$	$8.3478ms$	$31.842ms$	$251.943ms$
$MPI+Pthread+SSE+row$	$14.8326ms$	$62.520ms$	$496.729ms$
$MPI+Pthread+AVX+row$	$12.7629ms$	$35.840ms$	$257.238ms$

(i) 对于同一种策略（同为omp或者同为$Pthread$）使用AVX策略比SSE要更快

(ii)而且对于数据规模比较小时，使加速的差异并不是很大

②结果分析：

(i) 使用AVX_{256bit}一次能够处理8个$float$类型的数据，但是使用SSE_{128bit}一次只能处理4个 $float$类型的数据。所以使用AVX比SSE更快，但是并不能达到严格的二倍，因为在实际的使用过 程中开存在很多其他的时间开销（如调用，通信等）。

(ii) 在数据规模较小时，由于AVX_{256bit}代码量以及调度通信的开销更大，本身数据的计算时间很小，所以加速效果并不明显。

5.4 对比实验四：

目的： 探究任务的划分（行/列）对于 $MPI+omp/Pthread$程序的影响

（控制线程数等其他无关的变量，对于每一种不同的划分策略改变不同的数据规模，多次进行实验取平均值）

实验1：$MPI+omp+SSE+col$

实验2：$MPI+omp+SSE+row$

实验3：$MPI+Pthread+SSE+col$

实验4：$MPI+Pthread+SSE+row$

①实验结果：

数据规模	512	1024	2048
$MPI+omp+SSE+col$	$13.3279s$	$65.936ms$	$513.713ms$
$MPI+omp+SSE+row$	$10.2467ms$	$56.347ms$	$491.238ms$
$MPI+Pthread+SSE+col$	$19.6329ms$	$71.952ms$	$520.127ms$
$MPI+Pthread+SSE+row$	$14.8326ms$	$62.520ms$	$496.729ms$

(i) 横向对比发现 omp 的速度比 $Pthread$ 更快

(ii) 纵向对比发现，按照行的计算任务划分比按照列的任务划分速度更快

②结果分析：

(i) 由于 omp 是编译器自动创建新的子进程，而 $Pthread$ 是程序员手动创建新的进程，在创建的 时间开销上一定$t_{自动} < t_{手动}$，所以在速度上 omp 要比 $Pthread$ 更快。

(ii) 由于计算任务按列划分时，各个进程之间的通信要更少，在各个进程之间的通信代价要更小， 所以按照行划分的的速度要更快。

6.实验总结：

①编写 MPI 与 $Pthread$ 与 omp 相结合的程序需要耗费大量的时间，对于程序员来说，编程难度· 比较大。

②而且 MPI 与 $Pthread$ 与 omp 相结合的程序在出现bug进行调试的过程也需要花费大量的时间 和精力，比较难调试以及维护。

③MPI 与 $Pthread$ 与 omp 相结合的程序在程序的性能上远高于其他的普通程序，多次的加速优 化使得程序的运行速度极快。

④通过这次实验，我们发现了如下的规律：

- 使用 MPI 对普通的程序进行加速，加速比约为2
- 对于 omp 和 $Pthread$ 两种策略，omp 的速度始终要快于 $Pthread$
- 对于不同的任务划分，按列行分始终要略快于按列划分，因为按列划分的通信开销更大

⑤通过实验中设置的对比实验，我们从中实践并分析了这个学期以来我们学过的多种并行化策略， 对并行程序设计这门课程更加了解、真正的实践能力大幅度提升。