多线程编程作业

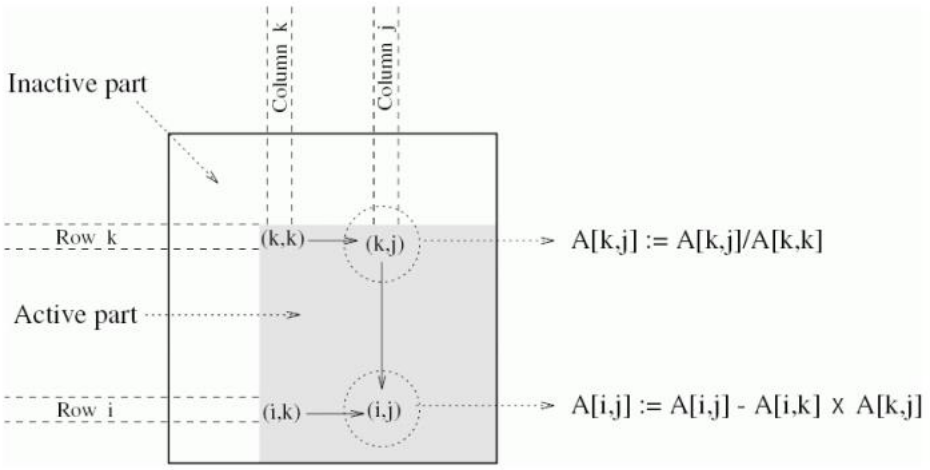
学号：2120210492　　姓名：肖飞

目录

一、问题描述

基于 X86/X64 平台完成普通高斯消去计算的基本 Pthread 并行化。

普通高斯消去法的计算模式如下图：



其中，主要涉及的操作为：在第 k 步时，对第 k 行从 (k, k) 开始进行除法操作，并且将后续的 k + 1 至 N 行进行减去第 k 行的操作。

二、实验环境

实验平台：x64 平台

CPU：12th Gen Intel(R) Core(TM) i5-12500H　　3.10 GHz

支持的指令集如下：

操作系统： Windows 11 系统

编译器：TDM-GCC 编译器

IDE： Code::Blocks 集成开发环境

三、多线程+SIMD 算法设计与实现

3.1、串行算法及其并行潜力分析

串行算法的伪代码如下（同 SIMD 作业）：

```
procedure LU (A)
begin
    for k := 1 to n do （外层循环）
        //除法操作
        for j := k+1 to n do（第一个内层循环）
            A[k, j ] := A[k, j]/A[k, k];
        endfor;
        A[k, k] := 1.0;

        //消去操作
```

```
        for i := k + 1 to n do  （第二个内层循环）
            for j := k + 1 to n do
                A[i, j ] := A[i, j ]  −  A[i, k]×A[k, j];
            endfor;
            A[i, k] := 0;
        endfor;
    endfor;
end LU
```

串行算法时间复杂度为 O(n^3)。共进行 n 轮消去步骤（外层循环），第 k 轮执行完第 k 行除法后（第一个内层循环），对于后续的 k + 1 至 n 行进行减去第 k 行的操作（第二个内层循环）。其中的主要数据依赖关系为，第二个内存循环中各行的消去操作依赖于第一个内层循环中除法操作的完成，需要通过同步操作确保一致性；而在第二个内存循环中，各行之间互不影响，没有相互的数据依赖关系，可采用多线程执行。

串行算法的具体实现如下：

```
//串行普通高斯消去算法
void m_gauss(int n)
{
    for(int k = 0 ; k < n ; k++)
    {
        for(int j = k+1 ; j < n ; j++)
        {
            m[k][j] = m[k][j]/m[k][k];
        }
        m[k][k] = 1.0;
        for(int i = k+1 ; i < n ; i++)
        {
            for(int j = k+1 ; j < n ; j++)
            {
                m[i][j] = m[i][j] - m[i][k] * m[k][j];
            }
            m[i][k] = 0;

        }
    }
}
```

完整代码实现请参考附件中的 GitHub 项目。

3.2、多线程+SIMD 算法并行优化设计与实现

本文主要使用课堂所学 pthread 编程任务划分和同步机制，针对消去部分

的两重循环，考虑通过不同编程策略或算法，分别设计并实现 pthread 多线程算法，同时考虑将其与 SIMD 算法结合。实验方面，改变矩阵的大小、线程数等参数，观测各算法运行时间的变化，对结果进行性能分析。

本文主要讨论的编程策略如下：

3.2.1、编程范式的选择

Pthread 编程有两种范式：静态线程和动态线程。

静态线程：程序初始化时创建好线程（池）。对于需要并行计算的部分，将任务分配给线程执行。但执行完毕后并不结束线程，等待下一个并行部分继续为线程分配任务。直至整个程序结束，才结束线程。优点是没有频繁的线程创建、销毁开销，性能更优；缺点是线程一直保持， 占用系统资源，可能造成资源浪费。

动态线程：在到达并行部分时，主线程才创建线程来进行并行计算，在这部分完成后，即销毁线程。在到达下一个并行部分时，再次重复创建线程——并行计算——销毁线程的步骤。优点是在没有并行计算需求时不会占用系统资源；缺点是有较大的线程创建和销毁开销。

就尽量避免频繁的创建和销毁线程来说，应该倾向于采用静态线程，如果每轮除法完成后创建线程，该轮消去完成后销毁线程。这样的话创建和销毁线程的次数过多，而线程创建、销毁的代价是比较大的。

如果采用静态线程范式，可以通过信号量同步、barrier 同步等同步方式避免频繁的创建和销毁线程。以信号量同步思路为例，主线程执行除法，工作线程执行消去。主线程开始时建立多个工作线程；在每一轮消去过程中，工作线程先进入睡眠，主线程完成除法后将它们唤醒，自己进入睡眠；工作线程进行消去操作，

完成后唤醒主线程，自己再进入睡眠；主线程被唤醒后进入下一轮消去过程， 直至任务全部结束销毁工作线程。

实际上上述方法存在程序逻辑复杂的问题，于是更好的一种方式是将多重循环都纳入线程函数中，消去操作对应的内层循环拆分，分配给工作线程。对于除法操作，可以只由一个线程执行，也可拆分由所有工作线程并行执行。除法和消去两个步骤后都要进行同步，以保证进入下一步骤（下一轮）之前上一步骤的计算全部完成，保证一致性。

本文后面会将两种范式都进行简单实现并进行对比。

3.2.2、不同任务划分策略与负载均衡

高斯消去过程中的计算主要集中在第一个内层循环（除法）和第二个内层循环（双重循环，消去），对应矩阵右下角 $(n-k+1)\times(n-k)$ 的子矩阵。因此，任务划分可以看作对此子矩阵的划分。

任务的划分方式总体上采用输入数据划分。而在第二个内层循环中，每行的消去操作计算量彼此相当，所以即可使用块划分，也可尝试自适应的动态任务划分方式。

另外，二维矩阵有行和列两个方向，故由此具体地又有水平划分与垂直划分两种方式。其中，对于除法部分，因为只涉及一行，只可能采用垂直划分（列划分）。而对于消去部分，即可采用水平划分（将其外层循环拆分，即每个线程分配若干行）， 也可采用垂直划分（将其内层循环拆分，即每个线程分配若干列）。两种划分策略在负载均衡上可能会有细微差异，在同步方面会有差异，cache 利用方面也会有不同。此外，在与 SIMD 结合时，SIMD 只能将行内连续元素的运

算打包进行向量化，即只能对最内层循环进行展开、向量化。

本文主要采取块划分方式，分别实现水平划分与垂直划分两种策略并进行比较，同时结合 SIMD 算法。

3.2.3、算法实现

综上所述，本文要实现的算法版本包括以下：

a. 串行算法版本

b. 动态线程版本

c. 静态线程+信号量同步，全部工作线程版本，水平划分版本

d. 静态线程+ barrier 同步，全部工作线程，垂直划分版本

以上除串行算法版本外，均与采用动态对齐策略的 SIMD 算法相结合，同时测试不同问题规模和不同线程数下的算法性能表现。

下面分别是 b、c 两个版本的伪代码。

动态线程版本：

线程数据结构以及线程函数：

```
typedef struct {
    int k; //消去的轮次
    int t_id; // 线程 id
}threadParam_t;

void *threadFunc(void *param) {
    threadParam_t *p = (threadParam_t*)param;
    int k = p -> k; //消去的轮次
    int t_id = p -> t_id; //线程编号
    int i = k + t_id + 1; //获取自己的计算任务

    For (int j = k + 1; j < n; ++j) do
        A[i][j] = A[i][j] - A[i][k] * A[k][j];
    end For
    A[i][k] = 0;
    pthread_exit(NULL);
}
```

主函数：

```
int main() {
    For (int k = 0; k < n; ++k) do
        //主线程做除法操作
        For (int j = k+1; j < n; j++) do
            A[k][j] = A[k][j] / A[k][k];
        end For
        A[k][k] = 1.0;

        //创建工作线程，进行消去操作
        int worker_count = n-1-k; //工作线程数量
        pthread_t* handles = Malloc();// 创建对应的 Handle
        threadParam_t* param = Malloc();// 创建对应的线程数据结构
        //分配任务
        For(int t_id = 0; t_id < worker_count; t_id++)
            param[t_id].k = k;
            param[t_id].t_id = t_id;
        end For
        //创建线程
        For(int t_id = 0; t_id < worker_count; t_id++)
            pthread_create();
        end For

        //主线程挂起等待所有的工作线程完成此轮消去工作
        For(int t_id = 0; t_id < worker_count; t_id++)
            pthread_join();
        end For
    end For

}
```

静态线程+信号量同步，全部工作线程，水平划分版本：

线程数据结构以及线程函数：

```
//线程数据结构定义
typedef struct {
    int t_id; //线程 id
}threadParam_t;

//信号量定义
sem_t sem_Divsion;
sem_t sem_Elimination;

//线程函数定义
void *threadFunc(void *param) {
    threadParam_t *p = (threadParam_t*)param;
    int t_id = p -> t_id;

    For (int k = 0; k < n; ++k) do
        // t_id 为 0 的线程做除法操作，其它工作线程先等待
        // 这里只采用了一个工作线程负责除法操作，同学们可以尝试采用多个工作线程完成除法操作
        // 比信号量更简洁的同步方式是使用 barrier
        if (t_id == 0)
            For (int j = k+1; j < n; j++) do
                A[k][j] = A[k][j] / A[k][k];
            end For
            A[k][k] = 1.0;
        else
            sem_wait(&sem_Divsion); // 阻塞，等待完成除法操作
        end if

        // t_id 为 0 的线程唤醒其它工作线程，进行消去操作
        if (t_id == 0)
            For (int t_id = 0; t_id < NUM_THREADS-1; ++t_id) do
                sem_post(&sem_Divsion);
            end For
        end if

        //循环划分任务（同学们可以尝试多种任务划分方式）
        For(int i=k+1+t_id; i < n; i += NUM_THREADS) do
            //消去
            For (int j = k + 1; j < n; ++j) do
                A[i][j] = A[i][j] -A[i][j] * A[k][j];
            end For;
            A[i][k]=0.0;
        end For;

        // 所有线程一起进入下一轮
        if(t_id == 0)
            For (int t_id = 0; t_id < NUM_THREADS-1; ++t_id) do

                sem_post(&sem_Elimination);
            end For
        else
            sem_wait(&sem_Elimination);
        end if
    end For;
    pthread_exit(NULL);
}
```

主函数：

```
int main() {
    //初始化信号量
    sem_init(&sem_Divsion, 0, 0);
    sem_init(&sem_Elimination, 0, 0);

    //创建线程
    pthread_t handles[NUM_THREADS];// 创建对应的 Handle
    threadParam_t* param[NUM_THREADS];// 创建对应的线程数据结构
    For(int t_id = 0; t_id < NUM_THREADS; t_id++)
        param[t_id].t_id = t_id;
        pthread_create();
    end For

    For(int t_id = 0; t_id < NUM_THREADS; t_id++) do
        pthread_join();
    end For

    sem_destroy(&sem_Divsion);
    sem_destroy(&sem_Elimination);

    return 0;
}
```

不同版本具体代码实现如下：

动态线程版本：

线程数据结构以及线程函数：

```
//动态线程版本数据结构
typedef struct {
    int k; //消去的轮次
    int t_id; // 线程 id
    int n; //问题规模
}threadParam_t_d;

//动态线程版本线程函数
void *threadFunc_d(void *param) {
    threadParam_t_d *p = (threadParam_t_d*)param;
    int k = p -> k; //消去的轮次
    int n = p -> n; //问题规模
    int t_id = p -> t_id; //线程编号
    int i = k + t_id + 1; //获取自己的计算任务

    __m128 vaik, vakj, vaij, vx;
    vaik = _mm_set_ps1(m[i][k]);
    int j;
    int start = k-k%4+4;
    for(j = k+1; j < start && j < n; j++){
        m[i][j] = m[i][j] - m[k][j]*m[i][k];
    }
    if(j != n){
        for(j = start; j+4 <= n; j+=4){
            vakj = _mm_load_ps(&m[k][j]);
            vaij = _mm_load_ps(&m[i][j]);
            vx = _mm_mul_ps(vakj, vaik);
            vaij = _mm_sub_ps(vaij, vx);
            _mm_store_ps(&m[i][j], vaij);
        }
        if(j < n){
            for(;j < n; j++){
                m[i][j] = m[i][j] - m[k][j]*m[i][k];
            }
        }
    }
    m[i][k] = 0;
    pthread_exit(NULL);
}
```

主函数：

```
//动态线程版本
void m_gauss_d(int n){
    for(int k = 0; k < n; ++k){
        //主线程做除法操作
        __m128 vt, va;
        vt = _mm_set_ps1(m[k][k]);
        int j;
        int start = k-k%4+4;
        for(j = k+1; j < start && j < n; j++){
            m[k][j] = m[k][j]/m[k][k];
        }
        if(j != n){
            for(j = start; j+4 <= n; j+=4){
                va = _mm_load_ps(&m[k][j]);
                va = _mm_div_ps(va, vt);
                _mm_store_ps(&m[k][j], va);
            }
            if(j < n){
                for(;j < n; j++){
                    m[k][j] = m[k][j]/m[k][k];
                }
            }
        }
        m[k][k] = 1.0;

        //创建工作线程，进行消去操作
        int worker_count = n-1-k; //工作线程数量
        pthread_t* handles = (pthread_t*)malloc(worker_count*sizeof(pthread_t));// 创建对应的 Handle
        threadParam_t_d* param = (threadParam_t_d*)malloc(worker_count*sizeof(threadParam_t_d));// 创建对应的线程数据结构
        //分配任务
        for(int t_id = 0; t_id < worker_count; t_id++){
            param[t_id].k = k;
            param[t_id].n = n;
            param[t_id].t_id = t_id;
        }

        //创建线程
        for(int t_id = 0; t_id < worker_count; t_id++){
            pthread_create(&handles[t_id], NULL, threadFunc_d, (void*)&param[t_id]);
        }

        //主线程挂起等待所有的工作线程完成此轮消去工作
        for(int t_id = 0; t_id < worker_count; t_id++){
            pthread_join(handles[t_id], NULL);
        }

        free(handles);
        free(param);
    }
}
```

静态线程+信号量同步，全部工作线程，水平划分版本：

线程数据结构以及线程函数：

```
//静态线程版本线程数据结构定义
typedef struct {
    int t_id; //线程 id
    int num_threads; //线程数
    int n; //问题规模
}threadParam_t;

//信号量定义
sem_t sem_Divsion;
sem_t sem_Elimination;
```

```
//静态线程+信号量同步，全部工作线程，水平划分版本线程函数定义
void *threadFunc_h(void *param) {
    threadParam_t *p = (threadParam_t*)param;
    int t_id = p -> t_id;
    int num_threads = p -> num_threads;
    int n = p -> n;
    __m128 vt, va, vaik, vakj, vaij, vx;

    for(int k = 0; k < n; ++k){
        // t_id 为 0 的线程做除法操作，其它工作线程先等待
        if (t_id == 0){
            vt = _mm_set_ps1(m[k][k]);
            int j;
            int start = k-k%4+4;
            for(j = k+1; j < start && j < n; j++){
                m[k][j] = m[k][j]/m[k][k];
            }
            if(j != n){
                for(j = start; j+4 <= n; j+=4){
                    va = _mm_load_ps(&m[k][j]);
                    va = _mm_div_ps(va, vt);
                    _mm_store_ps(&m[k][j], va);
                }
                if(j < n){
                    for(;j < n; j++){
                        m[k][j] = m[k][j]/m[k][k];
                    }
                }
            }
            m[k][k] = 1.0;
        }else{
            sem_wait(&sem_Divsion); // 阻塞，等待完成除法操作
        }

        // t_id 为 0 的线程唤醒其它工作线程，进行消去操作
        if (t_id == 0){
            for(int t_id = 0; t_id < num_threads-1; ++t_id){
                sem_post(&sem_Divsion);
            }
        }

        //循环划分任务
        for(int i=k+1+t_id; i < n; i += num_threads){
            //消去
            vaik = _mm_set_ps1(m[i][k]);
            int j;
            int start = k-k%4+4;
            for(j = k+1; j < start && j < n; j++){
                m[i][j] = m[i][j] - m[k][j]*m[i][k];
            }
            if(j != n){
                for(j = start; j+4 <= n; j+=4){
                    vakj = _mm_load_ps(&m[k][j]);
                    vaij = _mm_load_ps(&m[i][j]);
                    vx = _mm_mul_ps(vakj, vaik);
                    vaij = _mm_sub_ps(vaij, vx);
                    _mm_store_ps(&m[i][j], vaij);
                }
                if(j < n){
                    for(;j < n; j++){
                        m[i][j] = m[i][j] - m[k][j]*m[i][k];
                    }
                }
            }
            m[i][k] = 0.0;
        }

        // 所有线程一起进入下一轮
        if (t_id == 0){
            for(int t_id = 0; t_id < num_threads-1; ++t_id){
                sem_post(&sem_Elimination);
            }
        }else{
            sem_wait(&sem_Elimination);
        }
    }

    pthread_exit(NULL);
}
```

主函数:

```
//静态线程+信号量同步，全部工作线程，水平划分版本
void m_gauss_h(int n, int num_threads){
    //初始化信号量
    sem_init(&sem_Divsion, 0, 0);
    sem_init(&sem_Elimination, 0, 0);


    //创建线程
    pthread_t handles[num_threads];// 创建对应的 Handle
    threadParam_t param[num_threads];// 创建对应的线程数据结构
    for(int t_id = 0; t_id < num_threads; t_id++){
        param[t_id].t_id = t_id;
        param[t_id].n = n;
        param[t_id].num_threads = num_threads;
        pthread_create(&handles[t_id], NULL, threadFunc_h, (void*)&param[t_id]);
    }

    for(int t_id = 0; t_id < num_threads; t_id++){
        pthread_join(handles[t_id], NULL);
    }

    sem_destroy(&sem_Divsion);
    sem_destroy(&sem_Elimination);
}
```

静态线程+ barrier 同步，全部工作线程，垂直划分版本：

（线程数据结构同水平划分版本）线程函数：

```
//barrier定义
pthread_barrier_t barrier_Divsion;
pthread_barrier_t barrier_Elimination;
int num_blocks;//块划分大小

//静态线程+信号量同步，全部工作线程，垂直划分版本线程函数定义
void *threadFunc_v(void *param) {
    threadParam_t *p = (threadParam_t*)param;
    int t_id = p -> t_id;
    int num_threads = p -> num_threads;
    int n = p -> n;
    __m128 vt, va, vaik, vakj, vaij, vx;

    for(int k = 0; k < n; ++k){
        // t_id 为 0 的线程做除法操作，其它工作线程先等待
        if (t_id == 0){
            vt = _mm_set_ps1(m[k][k]);
            int j;
            int start = k-k%4+4;
            for(j = k+1; j < start && j < n; j++){
                m[k][j] = m[k][j]/m[k][k];
            }
            if(j != n){
                for(j = start; j+4 <= n; j+=4){
                    va = _mm_load_ps(&m[k][j]);
                    va = _mm_div_ps(va, vt);
                    _mm_store_ps(&m[k][j], va);
                }
                if(j < n){
                    for(;j < n; j++){
                        m[k][j] = m[k][j]/m[k][k];
                    }
                }
            }
            m[k][k] = 1.0;
            num_blocks = (n-k-1)/num_threads;

            if(num_blocks < 8){
                num_blocks = 8;
            }
        }

        // 阻塞，等待完成除法操作
        pthread_barrier_wait(&barrier_Divsion);
```

```
        //块划分任务
        int j = k+1+t_id*num_blocks;
        int o = j;
        if(j < n){
            int my_end = j+num_blocks;
            if(my_end > n){
                my_end = n;
            }

            if(t_id == num_threads-1){
                my_end = n;
            }

            int my_start = j-j%4+4;
            for(int i = k+1; i < n; i++){
                j = o;
                for(; j < my_start && j < my_end; j++){
                    m[i][j] = m[i][j] - m[k][j]*m[i][k];
                }

                if(j != my_end){
                    //消去
                    vaik = _mm_set_ps1(m[i][k]);
                    for(j = my_start; j+4 <= my_end; j+=4){
                        vakj = _mm_load_ps(&m[k][j]);
                        vaij = _mm_load_ps(&m[i][j]);
                        vx = _mm_mul_ps(vakj, vaik);
                        vaij = _mm_sub_ps(vaij, vx);
                        _mm_store_ps(&m[i][j], vaij);
                    }
                    if(j < my_end){
                        for(;j < my_end; j++){
                            m[i][j] = m[i][j] - m[k][j]*m[i][k];
                        }
                    }
                }
            }
        }


        // 所有线程一起进入下一轮
        pthread_barrier_wait(&barrier_Elimination);

        if (t_id == 1){
            for(int i = k+1; i < n; i++){
                m[i][k] = 0.0;
            }
        }

    }

    pthread_exit(NULL);
}
```

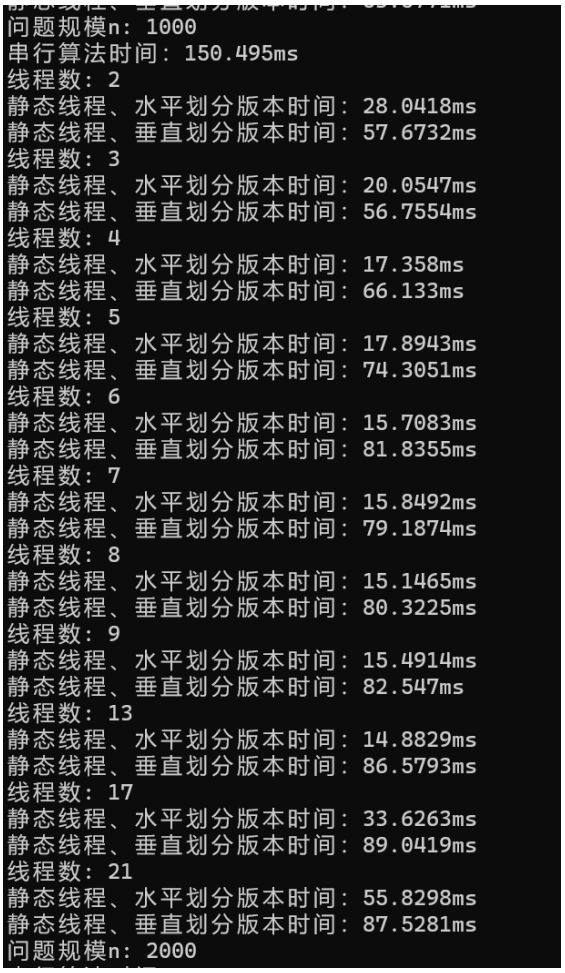
主函数:

```
//静态线程+信号量同步，全部工作线程，垂直划分版本
void m_gauss_v(int n, int num_threads){
    //初始化barrier
    pthread_barrier_init(&barrier_Divsion, NULL, num_threads);
    pthread_barrier_init(&barrier_Elimination, NULL, num_threads);

    //创建线程
    pthread_t handles[num_threads];// 创建对应的 Handle
    threadParam_t param[num_threads];// 创建对应的线程数据结构
    for(int t_id = 0; t_id < num_threads; t_id++){
        param[t_id].t_id = t_id;
        param[t_id].n = n;
        param[t_id].num_threads = num_threads;
        pthread_create(&handles[t_id], NULL, threadFunc_v, (void*)&param[t_id]);
    }

    for(int t_id = 0; t_id < num_threads; t_id++){
        pthread_join(handles[t_id], NULL);
    }

    pthread_barrier_destroy(&barrier_Divsion);
    pthread_barrier_destroy(&barrier_Elimination);
}
```

完整代码请参考附件中的 GitHub 项目。

四、实验及结果分析

程序不同问题规模下、不同算法/编程策略性能测试结果部分截图如下:



整理如下:

问题规模(n)	线程数	串行算法(ms)	动态线程(ms)	静态线程+水平划分(ms)	静态线程+垂直划分(ms)	问题规模(n)	线程数	串行算法(ms)	动态线程(ms)	静态线程+水平划分(ms)	静态线程+垂直划分(ms)
50	/		63.5297			100	/		198.604		
	1	0.0213					1	0.1587			
	2			0.2039	0.7294		2			0.1559	1.2764
	3			0.2337	1.069		3			0.1599	1.5884
	4			0.3326	1.6669		4			0.2381	1.6546
	5			0.2871	1.6119		5			0.2562	2.9013
	6			0.371	1.8319		6			0.3125	4.3205
	7			0.4026	1.896		7			0.4179	3.3305

	8			0.4334	2.14		8			0.4401	3.8631
	9			0.5249	2.463		9			0.5088	4.0313
400	/					700	/				
	1	10.3636					1	57.8316			
	2			2.7388	9.4892		2			11.8087	24.358
	3			2.3627	9.9521		3			8.6312	30.9187
	4			2.5492	12.63		4			7.0318	30.9332
	5			2.1879	17.2937		5			7.9386	37.5803
	6			2.2156	17.4481		6			7.8585	42.2696
	7			2.1525	17.9691		7			7.0446	43.0143
	8			2.8679	17.9594		8			7.7427	45.108
	9			6.65	18.2736		9			7.2485	46.4259
1000	/					2000	/				
	1	150.495					1	1203.47			
	2			28.0418	57.6732		2			261.89	323.437
	3			20.0547	56.7554		3			186.223	290.747
	4			17.358	66.133		4			138.319	283.545
	5			17.8943	74.3051		5			139.46	309.338
	6			15.7083	81.8355		6			127.02	315.205
	7			15.8492	79.1874		7			117.424	312.36
	8			15.1465	80.3225		8			104.525	320.395
	9			15.4914	82.547		9			99.0071	330.912
3000	/					4000	/				
	1	7265.5					1	17226.7			
	2			2779.2	2437.45		2			5765.67	5605.23
	3			2181.42	2067.05		3			5241.46	5059.61
	4			684.744	1391.18		4			4030.57	5096.63
	5			538.971	1534.59		5			2338.35	5144.74
	6			1103.95	1630.21		6			2994.36	4712.18
	7			453.471	1634.23		7			2022.44	4542.45
	8			745.91	1641.57		8			2245.77	4375.89
	9			903.358	2158.98		9			2494.29	5425.59

通过对以上结果进行分析，可以得到以下结论：

1、 当测试规模较小时，可能出现并行算法比串行算法更耗时的情况。

2、 一般地，线程数越多，获得的加速比越大，同时加速比还受具体硬件情况的影响而随线程数增多出现局部波动（如实验环境为 4P+8E，16 线程，当线程数增加到 4、8、16 附近时，加速比可能出现逆势减小）；结合了 SIMD 的多线程算法可以获得更高的加速比

（如线程数为 2 时，水平划分的策略加速比在较小问题规模下一般可达 5 左右，在较大规模时也能达到 3 左右，都超过了 2）。

3、 动态线程版本因为频繁创建销毁线程，开销过大。

4、 垂直划分策略的耗时一般大于水平划分，可能的因素包括 cache 空间局部性利用相对较差；每个垂直块划分可能都要动态对齐而使得 SIMD 加速相对不充分（水平划分每行只需要一次动态对齐），且线程数越多越不充分。

五、项目地址

Git 项目链接：https://github.com/XFLasdf/Guass_Pthread