并行程序设计与算法-期末项目

21307035 邓栩瀛

要求

分析不同并行编程框架下的矩阵乘法实现，了解不同并行框架在任务分配、通信、同步等编程特点及优化方式上的异同。

请适当给出并行编程的实现与优化的意见。

1.串行通用矩阵乘法及其优化

实现方法

```
vector<vector<double>> matrixMultiplication(const vector<vector<double>>& matrixA, const
vector<vector<double>>& matrixB) {
    int rowsA = matrixA.size();
    int colsA = matrixA[0].size();
    int colsB = matrixB[0].size();

    vector<vector<double>> result(rowsA, vector<double>(colsB));

    for (int i = 0; i < rowsA; i++) {
        for (int j = 0; j < colsB; j++) {
            for (int k = 0; k < colsA; k++) {
                result[i][j] += matrixA[i][k] * matrixB[k][j];
            }
        }
    }

    return result;
}
```

优化方法

（1）调整循环顺序

```
vector<vector<double>> matrixMultiplication(const vector<vector<double>>& matrixA, const
vector<vector<double>>& matrixB) {
    int rowsA = matrixA.size();
    int colsA = matrixA[0].size();
    int colsB = matrixB[0].size();

    vector<vector<double>> result(rowsA, vector<double>(colsB));

    for (int i = 0; i < rowsA; i++) {
        for (int k = 0; k < colsA; k++) {
            for (int j = 0; j < colsB; j++) {
                result[i][j] += matrixA[i][k] * matrixB[k][j];
            }
        }
    }
```

```
    return result;
}
```

（2）编译优化

```
gcc -g -O3 -fomit-frame-pointer -ffast-math main.cpp -lstdc++ -o main
```

-ffast-math: 关闭某些数学函数的严格准确性检查，以换取更高的性能，对浮点运算有帮助

-fomit-frame-pointer:用于决定是否在汇编代码中包含帧指针。帧指针是一个在函数调用时保存当前栈帧地址的寄存器。如果不使用帧指针，那么函数调用的开销会更小。

（3）循环展开

```
gcc -g -O3 -funroll-loops  main.cpp -lstdc++ -o main
```

-funroll-loops: 展开循环，以减少循环控制的开销，对循环嵌套的程序有性能提升作用。

（4）采用Intel MKL

```
void multiply_matrices(int m, int n, int k, double* A, double* B, double* C)
{
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, k, n, 1.0, A, n, B, k, 0.0, C,
k);
}
```

实验结果

版本	实验方法	运行时间(sec)
1	Python	231.433815
2	C/C++	5.91043
3	调整循环顺序	3.71451
4	编译优化	0.53627
5	循环展开	0.523683
6	Intel MKL	0.055911

使用高级语言（如Python）可能导致较长的运行时间，而使用低级语言（如C/C++）可以显著提升性能。对代码进行优化，如调整循环顺序、编译优化和循环展开，可以进一步提高性能。此外，使用专门的库（如Intel MKL）也可以显著提升性能。因此，选择适当的语言、进行代码优化和利用专门库都是提高性能的关键因素。

2.基于分布式内存的并行编程框架的矩阵乘法（MPI）

实现方法

核心代码

广播输入

```
    MPI_Bcast(&matrixSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&numProcesses, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

计算每个进程需要处理的矩阵行数

```
    int rowsPerProcess = matrixSize / numProcesses;
    int startRow = rank * rowsPerProcess;
    int endRow = (rank == numProcesses - 1) ? matrixSize : (startRow + rowsPerProcess);
```

分配内存

```
    int** matrixA = new int*[matrixSize];
    int** matrixB = new int*[matrixSize];
    int** result = new int*[matrixSize];
    for (int i = 0; i < matrixSize; i++) {
        matrixA[i] = new int[matrixSize];
        matrixB[i] = new int[matrixSize];
        result[i] = new int[matrixSize];
    }
```

生成随机矩阵

```
    if (rank == 0) {
        generateRandomMatrix(matrixA, matrixSize);
        generateRandomMatrix(matrixB, matrixSize);
    }
```

广播随机矩阵

```
    for (int i = 0; i < matrixSize; i++) {
        MPI_Bcast(matrixA[i], matrixSize, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(matrixB[i], matrixSize, MPI_INT, 0, MPI_COMM_WORLD);
    }
```

每个进程分配计算任务

```
    int localRows = endRow - startRow;
    int** localResult = new int*[localRows];
    for (int i = 0; i < localRows; i++) {
        localResult[i] = new int[matrixSize];
    }
```

并行计算矩阵乘法

```
    double startTime = MPI_Wtime();
    matrixMultiply(matrixA, matrixB, localResult, matrixSize, startRow, endRow);
```

收集计算结果

```
    MPI_Gather(localResult[0], localRows * matrixSize, MPI_INT, result[0], localRows *
matrixSize, MPI_INT, 0, MPI_COMM_WORLD);
```

实验结果

运行时间，单位：秒

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.072771	0.321411	1.34084	6.81646	time out
2	0.070127	0.276058	1.18605	5.46334	54.2342
4	0.066944	0.267516	1.0983	4.7156	35.033
8	0.064895	0.256045	1.05402	4.3564	25.5674
16	0.067656	0.255371	1.04294	4.19303	20.832

1. 随着矩阵规模的增加，运行时间也随之增加。
2. 随着进程数的增加，总体上运行时间呈现下降的趋势，这说明并行计算可以有效地减少矩阵运算的时间消耗。然而，在某些情况下，增加进程数可能会引入额外的开销，导致运行时间反而增加。
3. 对于较小的矩阵规模（如128和256），增加进程数对运行时间的影响并不明显，这是因为较小的矩阵规模可能无法充分利用并行计算的优势，并且并行化引入的通信和同步开销可能抵消了并行计算的性能提升。
4. 随着进程数的增加，性能的提升逐渐减小。从1到2个进程时，性能提升较为显著（如128规模的矩阵），但增加更多的进程时，性能提升幅度逐渐减小，这是因为并行计算的效果受限于硬件资源和并行算法的特性。

优化建议

1.通信优化：

- 重叠通信与计算：在计算时启动通信，以隐藏通信延迟。
- 非阻塞通信：利用MPI的非阻塞通信函数（如`MPI_Isend`和`MPI_Irecv`）进行异步数据传输。

2.负载均衡：确保各个计算节点的计算负载均衡，避免某些节点成为瓶颈。

3.基于共享内存的CPU多线程编程（Pthreads）

实现方法

```
    pthread_t threads[MAX_THREADS]; //存储线程的标识符
    ThreadArgs thread_args[MAX_THREADS];//存储每个线程的参数

    for (int i = 0; i < num_threads; ++i) //循环创建多个线程
    {
        thread_args[i].thread_id = i;
        thread_args[i].num_threads = num_threads;
        thread_args[i].A = &A;
        thread_args[i].B = &B;
        thread_args[i].C = &C;
        pthread_create(&threads[i], NULL, multiply, (void*)&thread_args[i]);
    }

    for (int i = 0; i < num_threads; ++i) //循环等待每个线程执行完毕
    {
        pthread_join(threads[i], NULL);
```

```
    }
```

实验结果

运行时间，单位：秒

线程数	矩阵规模				
	128	256	512	1024	2048
1	0.018121	0.126202	0.879359	6.58175	112.664
2	0.020349	0.147105	0.887256	6.6904	119.91
4	0.020277	0.147617	0.942781	6.80991	133.562
8	0.023345	0.15305	1.14385	8.43453	183.479
16	0.021787	0.168542	1.10103	8.41133	139.031

1. 随着矩阵规模的增加，运行时间也随之增加，因为大型矩阵乘法需要更多的计算时间。
2. 在每个矩阵规模下，随着线程数量的增加，运行时间并不总是线性减少，可能是因为线程创建和管理带来了额外的开销，并且矩阵乘法的计算本身可能存在一定的串行性质。
3. 对于较小的矩阵（128x128），增加线程数量并没有显著改善性能，可能是因为矩阵太小，无法充分利用多线程的优势；当矩阵规模增大到2048x2048时，16个线程的性能反而略有上升。

优化建议

1.尽量减少互斥锁的使用，使用原子操作或减少共享数据访问。

2.将线程绑定到特定的CPU核心上，以减少线程迁移带来的开销。

4.GPU多线程编程CUDA

实现方法

kernel函数

```
__global__ void matrixMulKernel(float* A, float* B, float* C, int m, int n, int k) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < m && col < k) {
        float value = 0;
        for (int e = 0; e < n; ++e) {
            value += A[row * n + e] * B[e * k + col];
        }
        C[row * k + col] = value;
    }
}
```

矩阵初始化

```
void initializeMatrix(float* matrix, int rows, int cols) {
    for (int i = 0; i < rows * cols; ++i) {
        matrix[i] = static_cast<float>(rand()) / RAND_MAX;
    }
}
```

矩阵乘法

```
void matrixMulCUDA(int m, int n, int k, int blockSize) {
    size_t sizeA = m * n * sizeof(float);
    size_t sizeB = n * k * sizeof(float);
    size_t sizeC = m * k * sizeof(float);

    // 分配主机内存
    float* h_A = (float*)malloc(sizeA);
    float* h_B = (float*)malloc(sizeB);
    float* h_C = (float*)malloc(sizeC);

    // 初始化矩阵A和B
    initializeMatrix(h_A, m, n);
    initializeMatrix(h_B, n, k);

    // 分配设备内存
    float* d_A; cudaMalloc(&d_A, sizeA);
    float* d_B; cudaMalloc(&d_B, sizeB);
    float* d_C; cudaMalloc(&d_C, sizeC);

    // 将主机内存数据拷贝到设备内存
    cudaMemcpy(d_A, h_A, sizeA, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, sizeB, cudaMemcpyHostToDevice);

    // 定义CUDA网格和块结构
    dim3 blockDim(blockSize, blockSize);
    dim3 gridDim((k + blockSize - 1) / blockSize, (m + blockSize - 1) / blockSize);

    // 记录开始时间
    auto start = std::chrono::high_resolution_clock::now();

    // 调用矩阵乘法核函数
    matrixMulKernel<<<gridDim, blockDim>>>(d_A, d_B, d_C, m, n, k);

    // 同步设备
    cudaDeviceSynchronize();

    // 记录结束时间
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<float, std::milli> duration = end - start;

    // 将结果从设备内存拷贝回主机内存
    cudaMemcpy(h_C, d_C, sizeC, cudaMemcpyDeviceToHost);

    // 输出计算时间
    std::cout << "Running time: " << duration.count() << " ms" << std::endl;

    // 释放内存
    free(h_A);
```

```
    free(h_B);
    free(h_C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

实验结果

运行时间，单位：秒

线程块大小	矩阵规模				
	128	256	512	1024	2048
1	0.000655	0.003588	0.024682	0.217280	1.127790
2	0.000069	0.001138	0.008838	0.074999	0.396947
4	0.000085	0.000492	0.003701	0.028831	0.192990
8	0.000057	0.000267	0.001908	0.014485	0.115038
16	0.000043	0.000186	0.001219	0.009058	0.071836

1.随着线程块大小的增加，矩阵计算时间通常减少，但矩阵规模越小时，影响效果更显著。

2.线程块大小对于不同矩阵规模的影响不同，矩阵规模较小时，矩阵计算时间的降低幅度更大。

优化建议

1.选择合适的线程块大小，充分利用GPU的计算资源。

2.利用共享内存加速数据访问，减少全局内存访问的延迟。

3.减少寄存器使用，避免寄存器溢出到局部内存。

对比分析

编程框架	实现难度	并行性	内存管理	计算资源利用效率	调试难度
串行实现	低	无	简单	低	低
分布式内存并行MPI	高	高	需要显式管理，数据通信	高	高，涉及网络通信
共享内存并行Pthreads	高	高	需要显式管理，可能发生竞争	高	高，需处理同步问题
GPU多线程并行CUDA	高	很高	需要显式管理，优化显存使用	很高	高，涉及CUDA调试工具

总结

1.串行实现适合小规模数据，优化侧重于缓存和向量化。

2.MPI适合大规模分布式计算，优化侧重于通信和负载均衡。

3.Pthreads/OpenMP适合共享内存多核系统，优化侧重于同步和线程调度。

4.CUDA适合大规模并行计算，优化侧重于内存访问和线程块配置。

特点/编程框架	串行实现	分布式内存并行（MPI）	共享内存并行（Pthreads/OpenMP）	GPU多线程并行（CUDA）
任务分配	单一处理器处理整个任务	按行、列或块划分任务到各个节点	按行、列或块划分任务到各个线程	按块划分任务到各个线程块
通信	无需通信	使用消息传递进行数据交换（MPI_Send/Recv）	共享内存模型下无需显式通信	主机与设备之间的数据传输（cudaMemcpy）
同步	无需同步	使用阻塞/非阻塞通信函数进行同步	使用锁、屏障等同步机制	使用同步函数（__syncthreads）
缓存优化	数据局部性优化	分块以适应缓存，减少通信带来的缓存未命中	分块以适应缓存，减少共享内存访问冲突	使用共享内存，加速局部数据访问
向量化	SIMD指令集	依赖于各节点的处理器向量化能力	依赖于多核CPU的向量化能力	使用GPU的向量处理单元（Warp）
通信优化	无需通信	重叠通信与计算，使用非阻塞通信	无需显式通信	优化主机与设备的内存传输，减少传输次数
同步优化	无需同步	尽量减少通信同步开销，使用异步通信	减少锁使用，采用无锁编程技术	优化线程块内同步，减少全局同步使用
负载均衡	无需负载均衡	均衡分配任务以避免节点负载不均	动态/静态调度策略，线程绑定优化	适当配置线程块和网格尺寸，均衡GPU负载
线程/进程调度	无需调度	MPI进程调度依赖系统调度	动态/静态调度，线程绑定到CPU核心	线程块与线程的调度由CUDA运行时管理
内存对齐	无显式要求	数据划分时考虑内存对齐	数据结构设计时考虑内存对齐	优化数据结构以确保内存对齐，提高访问效率