基于 OpenMP 并行加速矩阵乘法

刘沛灵

摘要：在大型科学计算中，矩阵乘法是一项计算密集型任务，广泛应用于工程数值计算。传统的串行计算方法在处理这些运算时，因计算时间过长和效率低下，已难以满足实际需求，为了解决这一问题，需要不断探索适用于并行计算的矩阵乘法方法。与串行算法相比，矩阵乘法的并行算法涉及更多复杂因素。本文利用 OpenMP、MPI 等 API 进行多核并行计算，将矩阵按照一定规则进行分块处理，每个部分负责计算一个矩阵块，而且利用局部性原理从而显著缩短了计算时间。这种方法有效地提升了计算效率，将原本耗时的运算大幅加速。同时，我们也可以考虑矩阵的特殊情况—稀疏矩阵，对于稀疏矩阵可以采取特殊并行算法进行计算，可以大大提高计算效率。

关键词：矩阵乘法；Strassen 算法；OpenMP；MPI；稀疏矩阵

Parallel Acceleration of Matrix Multiplication Based on OpenMP

Liu Peiling

Abstract:In large-scale scientific computing, matrix multiplication is a computationally intensive task widely used in engineering numerical calculations. Traditional serial computation methods, due to their long computation times and low efficiency, have become inadequate for meeting practical needs. To address this issue, it is essential to continuously explore parallel computation methods for matrix multiplication. Compared to serial algorithms, parallel algorithms for matrix multiplication involve more complex factors. This paper utilizes APIs such as OpenMP and MPI for multi-core parallel computing, partitioning the matrix according to specific rules, where each part is responsible for computing a matrix block. By leveraging the principle of locality, this approach significantly reduces computation time. This method effectively enhances computational efficiency, greatly accelerating previously time-consuming calculations. Additionally, we can consider special cases of matrices, such as sparse matrices. For sparse matrices, specialized parallel algorithms can be employed, which can significantly improve computational efficiency.

Key words:Matrix Multiplication; Strassen Algorithm; OpenMP; MPI; Sparse Matrix

1 引言

在科学计算和工程数值计算中，矩阵乘法是一种非常常见且计算量巨大的运算。传统的串行计算方式在处理大规模矩阵乘法时，往往面临计算时间长、效率低的问题，难以满足日益增长的计算需求。通过并行化矩阵乘法，可以显著提高计算效率，减少计算时间，具有重要的意义。例如，在需要实时处理和响应的应用场景中，图像处理、信号处理和金融分析等，并行计算能够提供快速的计算结果，支持实时应

用的需求。同时，高效的并行计算能力使得科学研究人员能够进行更复杂的模拟和分析，推动了科学研究的进步。与此同时，随着处理器核心数的增加和串行编程复杂度的提高，如何利用多核处理器编写可并行的程序以提高计算机的处理能力开始渐渐引起了人们的关注，通过并行的方式，可以大大减少串行程序的原执行时间（加速比与核心数有关）。简单来说，通过合理的并行算法设计和高效的并行计算实现，可以显著提升计算能力，推动各领域的发展。本文基于 ubuntu20.04 的系统，设计并实现了并行矩阵相乘算法，对比串行程序与并行程序之间的加速比，时间复杂度和空间复杂度等，研究了一核、二核、四核以及八核处理器在并行程序下的运算处理能力，为大规模矩阵处理的并行化提供了研究基础。

2 基于 OMP 的矩阵计算编程模型

2.1 OpenMP 简介

OpenMP（Open Multi-Processing）是一种用于多平台共享内存并行编程的 API（应用程序接口），主要用于 C、C++和 Fortran 编程语言。它提供了一套编译器指令、库函数和环境变量，使程序员能够简单高效地开发并行应用程序。OpenMP 的并行编程指导语句以#pragma omp 开始，后面跟具体的功能指令（或命令）。程序启动时，只有一个主线程。当执行到并行区域时，主线程会创建多个工作线程，这些线程共同执行并行区域的代码（如图 1）。具有工作分配、数据共享和私有化和同步制等特性。

2.2 串行程序算法原理

该算法原理由线性代数中基础的矩阵相乘方法得到。将一个矩阵用一个二维数组表示，则一个 N×M 的矩阵有 N 行 M 列元素，两矩阵 A，B 相乘，乘积 C 的元素为 $C_{i,j}\ \ (0<=i<=N-1, 0<=j<=M-1)$ 可由下式得到:

$$C_{i,j}=\sum_{k=0}^{N-1} a_{i,k}b_{k,j}$$

矩阵 A 的第 n 行的各元素与矩阵 B 的第 m 列的各元素分别相乘，并将乘积相加得到矩阵 C 的第 n 行第 m 列的一个元素 $C_{n,m}$。

关键代码如下：

```
//A 的行数
for (int i = 0; i < A_ROWS; ++i) {
//B 的列数
  for (int j = 0; j < B_COLS; ++j) {
      C[i][j] = 0;
        for (int k = 0; k < A_COLS; ++k) {
            C[i][j] += A[i][k] * B[k][j];
          }
        }
    }
```

根据题目要求，此处矩阵 A 为 2048*4096，矩阵 B 为 4096*2048，故时间复杂度为 O(2048*4096*2048)，简单来看，即$O(n^3)$。

2.3 并行程序算法原理

上文已经提到矩阵相乘串行算法的时间复杂度为 T(M,N,L)=O(M×N×L)，M，N 为矩阵 A 的行数和列数，N，L 为矩阵 B 的行数和列数。本文在该算法的基础上，进行了一定改进，能够实现并行计算矩阵。基本思想是将要计算的矩阵 C[M][N]进行块的划分，将其合理地分配为均匀的四块，于是可以交给 4 个线程分别执行。即如下述公式：

$$C[M_1][N_1]=\sum A[M_1][k]*B[k][L_1]$$
$$C[M_1][N_2]=\sum A[M_1][k]*B[k][L_2]$$
$$C[M_2][N_1]=\sum A[M_2][k]*B[k][L_1]$$
$$C[M_2][N_2]=\sum A[M_2][k]*B[k][L_2]$$

矩阵 C 的分块方法是，将矩阵 A 的行数的一半作为中分点，矩阵 B 的列数的一半作为中分点，两两相切则可以保证四块的计算量近乎一致(如图 2)，每个线程的负载一致，可以保证四个线程可以在接近相

同的时间完成计算。

图1

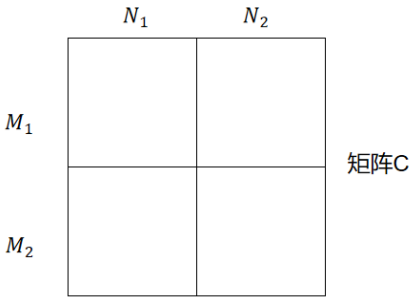


图2

完成分块后即可使用 OpenMP 编程语言实现矩阵相乘的并行算法，核心代码如下：

```
    //分配四个线程，并将矩阵的四个部分分配给这四个线程进行计算
omp_set_num_threads(4);
#pragma omp parallel for private(i, j, k)
for (int block = 0; block < 4; ++block) {
   //判断是哪一部分的矩阵计算
int row_start = (block / 2) * block_rows;
int col_start = (block % 2) * block_cols;
//确定矩阵位置并进行矩阵相乘计算
for (i = row_start; i < row_start + block_rows; ++i) {
      for (j = col_start; j < col_start + block_cols; ++j) {
            double sum = 0;
              for (k = 0; k < A_COLS; ++k) {
                sum += A[i][k] * B[k][j];
                                }
              C[i][j] = sum;
            }
        }
}
```

从代码中可以窥见，将 i, j, k 声明为每个线程的私有变量。四个线程之间存在共享读，由于每个线程只操作 c 矩阵中不同的部分，并且所有可能引起数据冲突的变量都被正确地声明为私有变量，所以不会发生数据冲突。这段代码在分配矩阵块和并行计算上是安全的，不会引起未定义行为。

2.4 矩阵乘法并行与非并行性能比较

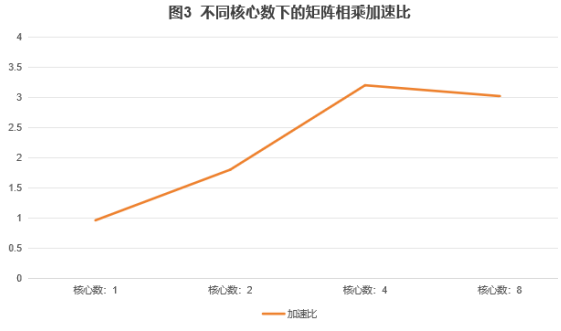
矩阵相乘问题规模为：矩阵 A:2048*4096,矩阵 B：4096*2048。记录串行时间与并行时间如表 1 所示，其中环境配置为：ubuntu20.04，四核，内存大小 4G，四线程。

表1 矩阵乘法执行的时间（单位 s）

时间	第一次	第二次	第三次	第四次	第五次	平均时间
串行	127	131	132	136	129	131
OpenMP	38	41	42	43	40	40.8

根据实验的结果分析可得：并行算法所花费的时间相较串行算法的时间，减少了 2/3，可以发现 4 个线程并行并不能保证所花时间减小为原来 1/4，即加速比为 4，这是因为启动线程和管理线程的并行化过程本身会引入一些开销。这些开销包括线程的创建、调度、同步以及数据分割和合并的时间。这些额外的开销会减少并行化带来的性能提升。因此我们可以根据实际硬件环境优化线程数量，通常建议线程数与物理 CPU 核数相匹配，以避免过多的线程上下文切换。

为了更加明显发现核心数与线程数之间的比例对程序计算的影响，我记录了不同核心数下，四个线程的程序的运行结果，如图 3。



可以发现在线程数为 4 的情况下，并行化可以显著提高矩阵相乘的计算性能，但加速比并不是线性增长的。随着核心数的增加，性能提升会逐渐趋于平缓甚至下降，这主要是由于并行化开销、内存带宽限制和负载不均衡等因素导致的。因此可以说，在核心数和线程数一致时可以得到一个较好的效果。

2.5 Strassen 并行算法设想

Strassen 矩阵乘法算法是由 Volker Strassen 在 1969 年提出的，是第一个被发现的时间复杂度低于传统矩阵乘法（$O(n^3)$）的算法。Strassen 算法将两个大矩阵分解为更小的矩阵，通过递归地进行这些更小矩阵的乘法，最终减少总体乘法操作次数。Strassen 算法的时间复杂度为 $O(n^{\log_2 7}) \approx O(n^{2.81})$，优于传统的 $O(n^3)$。

步骤如下：

假设我们有两个 2×2的矩阵 A 和 B：

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

$$B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

传统方法需要 8 次乘法，而 Strassen 算法只需要 7 次乘法。具体步骤如下：

1.

计算中间矩阵：

$M_1 = (a_{11} + a_{22})(b_{11} + b_{22})$
$M_2 = (a_{21} + a_{22})b_{11}$
$M_3 = a_{11}(b_{12} - b_{22})$
$M_4 = a_{22}(b_{21} - b_{11})$
$M_5 = (a_{11} + a_{12})b_{22}$
$M_6 = (a_{21} - a_{11})(b_{11} + b_{12})$
$M_7 = (a_{12} - a_{22})(b_{21} + b_{22})$

2.

组合结果：

$C_{11} = M_1 + M_4 - M_5 + M_7$
$C_{12} = M_3 + M_5$
$C_{21} = M_2 + M_4$
$C_{22} = M_1 - M_2 + M_3 + M_6$

如此可以设计代码如下：

```
//add_matrix 为矩阵加法，sub_matrix 为矩阵减法
        // 计算 M1 到 M7
        add_matrix(A11, A22, temp1, m);
        add_matrix(B11, B22, temp2, m);
        strassen(temp1, temp2, M1, m);

        add_matrix(A21, A22, temp1, m);
        strassen(temp1, B11, M2, m);

        sub_matrix(B12, B22, temp2, m);
        strassen(A11, temp2, M3, m);

        sub_matrix(B21, B11, temp2, m);
        strassen(A22, temp2, M4, m);

        add_matrix(A11, A12, temp1, m);
        strassen(temp1, B22, M5, m);

        sub_matrix(A21, A11, temp1, m);
        add_matrix(B11, B12, temp2, m);
        strassen(temp1, temp2, M6, m);

        sub_matrix(A12, A22, temp1, m);
        add_matrix(B21, B22, temp2, m);
        strassen(temp1, temp2, M7, m);

        // 组合结果
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < m; ++j) {
                C11[i][j] = M1[i][j] + M4[i][j] - M5[i][j] + M7[i][j];
                C12[i][j] = M3[i][j] + M5[i][j];
                C21[i][j] = M2[i][j] + M4[i][j];
                C22[i][j] = M1[i][j] - M2[i][j] + M3[i][j] + M6[i][j];
            }
        }
```

在本文中可以将矩阵 A 看为 $A = (A_1 \quad A_2)$，将矩阵 B 看为$B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$。那么 AB 相乘的结果为$C = (A_1B_1 \quad A_2B_2)$。至此，有了本文设想，将 A，B 分别都分为两个方阵，即 2024*2024，将A_1B_1和A_2B_2分别分配给两个线程采用 strassen 算法进行计算。从理论的时间复杂度上来讲，此方法应比前文采用两个线程并行计算的加速比更高一点，但根据记录数据（如表 2）来看，采用该方法甚至比串行时间更慢。根据其代码结构可以发现 Strassen 算法通过递归分解矩阵进行计算。对于大规模矩阵，递归深度会很深，且都会调用函数，导致每次递归调用的开销累积，最终显著增加计算时间。然而，不可忽视的是，若能解决其开销问题，在时间复杂度上来看，该算法相比于传统算法具有优势，因此，这也是今后可以努力的一个方向。

表2 矩阵乘法执行的时间（单位 s）

时间	第一次	第二次	第三次	平均时间
串行	130	142	162	144
strassen	443	432	431	435

除此之外使用 FFT 进行矩阵乘法也是一种高效的算法，通过将矩阵乘法转换为频域的乘法运算，可以显著减少计算时间。虽然实现复杂度较高，但在大规模矩阵乘法和卷积运算中，FFT 矩阵乘法具有显著的优势。

所以在对矩阵乘法并行化计算时，可以首先降低矩阵乘法本身的算法复杂度，再与 OpenMP 并行语言结合，在某些情况下可以得到一个更好的结果。

3 编译优化设计及 MPI 并行计算

3.1 编译优化操作

GCC 提供多个优化级别，每个级别都会启用一组不同的优化，以在编译时平衡编译时间和运行时性能。例如，“-O0”指不进行任何优化，但调试信息最完整；“-O1”启用基本的优化，但运行时性能有所提升；“-O2”启用比”-O1“更多的优化，进一步提高运行时性能；“-O3”启用所有的-O2 优化，并且启用更多的高成本优化。除此之外还有“-march=native”等优化选项进行选择，本文的 OpenMP 并行算法代码的 Makefile 文件中选择了“-O3”优化，启用的优化包括：所有“-O2”优化、自动向量化、循环展开等，其本身优化就有向量化操作，但为了尝试向量化选项，我仍添加了“-ftree-vectorize”和“-march=native”。同时记录下了不同优化下程序运行所需要的时间，且在“-O3”的选项添加，对比不同情况的运行时间，如表 3。

表3 矩阵乘法执行的时间（单位 s）

时间	-O0	-O1	-O2	-O3	-O3 -ftree-vectorize -march=native
串行	220	129	124	127	125
并行	57	43	40	38	40

通过这些优化选项，可以根据需要在编译时间、代码大小和运行时性能之间取得平衡。在“O1”优化的情况下，所取得的结果与最优优化的情况差距不大，且“O1”相较于“O2”、“O3”等来说，占用的资源和编译时间更少，故此处采用“-O1”就可以得到一个不错的结果。

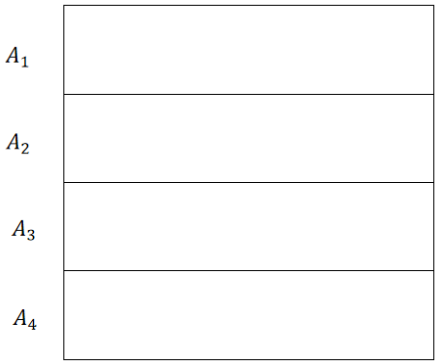
3.2 MPI 简介

MPI（Message Passing Interface，消息传递接口）是一个标准化的、可移植的消息传递库接口，广泛用于高性能计算（HPC）领域。MPI 中的并行任务是通过多个进程来实现的，每个进程都有自己的独立内存空间，同时，MPI 提供了进程间通信的机制，使得在分布式内存环境下的并行计算变得更加高效和便捷。调用这些接口，链接相应平台上的 MPI 库，就可以实现基于消息传递的并行计算，例如 MPI_Init、MPI_Comm_rank 等。

3.3 基于 MPI 的矩阵乘法

此处与上文介绍的矩阵乘法相似，将矩阵 A 分为四块(如图 4)，四等分后分别与矩阵 B 相乘得到矩阵 C。A_1、A_2、A_3、A_4 分别交给四个进程与矩阵 B 进行相乘，均分可以保证四个进程在接近的时间内完成计算。同时需要注意将矩阵 A、B 的二维数组化为一维数组后才能利用 MPI 的“MPI_Scatter”将矩阵 A 平均分配给四个进程。

图4



矩阵A

矩阵相乘核心代码如下：

```
        // 并行计算局部 C 矩阵
for (int i = 0; i < rows_per_proc; ++i) {
    for (int j = 0; j < B_COLS; ++j) {
        local_C[i * B_COLS + j] = 0;
        for (int k = 0; k < A_COLS; ++k) {
            local_C[i * B_COLS + j] += local_A[i * A_COLS + k] * B_linear[k * B_COLS + j];
                    }
                }
            }

            // 合并，收集局部 C 矩阵到进程 0
MPI_Gather(local_C, rows_per_proc * B_COLS, MPI_DOUBLE, C_parallel_linear, rows_per_proc * B_COLS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
// 进程 0 将线性化的 C_parallel 转换为二维数组
    if (rank == 0) {
        for (int i = 0; i < A_ROWS; ++i) {
            for (int j = 0; j < B_COLS; ++j) {
    C_parallel[i][j] = C_parallel_linear[i * B_COLS + j];
                }
            }
        }
```

首先计算每个进程所负责的局部矩阵 local_C。每个进程处理矩阵 A 的部分行和完整的矩阵，然后使用 MPI_Gather 将所有进程计算得到的局部矩阵 local_C 合并到进程 0 的，最后进程 0 将一维数组 C_parallel_linear 转换回二维矩阵 C_parallel 以进行后续处理或结果验证。

3.5 串行优化

优化缓存命中率的原理主要是通过改进数据访问模式，使得数据能够更高效地被加载到缓存中，从而减少缓存未命中的次数。其中利用到了局部性原理。空间局部性：当程序访问一块内存区域时，附近的内存位置也很可能会被访问。为了提高空间局部性，程序应尽量顺序访问数组中的元素。时间局部性：程序对某个内存位置的访问后，稍后的访问也很可能会访问到相同位置。在矩阵乘法中，多次访问同一行或同一列的元素会增加时间局部性。在计算一个小块（子矩阵）的乘法时，这些数据通常都在缓存中，减少了对主内存的访问。块化可以提高对数据的缓存局部性，从而减少缓存未命中的次数。

本文中，其使用 BLOCK_SIZE（代码里为 BLOCK_SIZE=32）作为块的大小，采用 OMP 作为并行算法，使用了四个线程，每次对 BLOCK_SIZE*BLOCK_SIZE 的区域进行计算，其并不会马上计算出矩阵 C 对应位置的值，而是根据每次块的值进行加和，其核心代码如下：

```
for (i = row_start; i < row_start + BLOCK_SIZE; ++i) {
    for (j = col_start; j < col_start + BLOCK_SIZE; ++j) {
        double sum = 0;
        for (k = k_start; k < k_start + BLOCK_SIZE; ++k) {
            sum += A[i][k] * B[k][j];
                                    }
        C[i][j] += sum;
                                }
    }
```

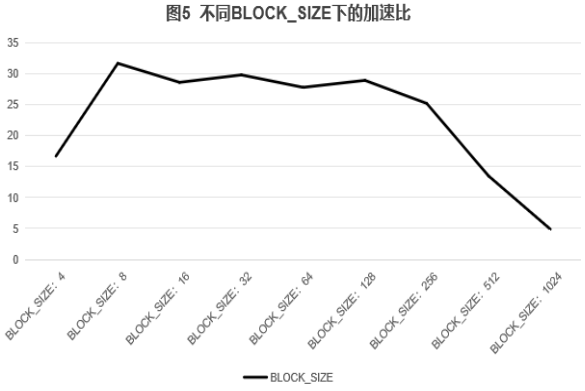
代码中通过块化的方式进行矩阵乘法，确保计算过程中数据局部性和缓存利用率。通过这些优化措施，可以显著提高矩阵乘法的性能，尤其是在处理大规模矩阵时。

此处矩阵相乘问题规模为：矩阵A:2048*4096,矩阵B：4096*2048。其中本地环境配置为：ubuntu20.04，四核，内存大小4G，四线程，记录不同大小的BLOCK_SIZE的耗时如表4。

表4 矩阵乘法执行的时间（单位 s）

BLOCK_SIZE	4	8	16	32	64	128	256	512	1024
串行	149.5	157.4	157.5	143.2	143.1	144.6	156.8	156.7	161.6
Open MP	9	4.97	5.5	4.8	5.15	5	6.2	11.6	33

计算出加速比如图5:



由图可以看出在 BLOCK_SIZE 较小（如 4 或 8）时，加速比较低，但随着 BLOCK_SIZE 增大，加速比逐渐提高，在某些 BLOCK_SIZE（如 16 和 32）达到最大值。

当 BLOCK_SIZE 进一步增大时，加速比开始下降，尤其是 BLOCK_SIZE 超过 128 以后，加速比显著降低。过小或过大的块大小都会导致性能下降：过小的块由于计算不充分和通信开销较大，过大的块由于缓存溢出和同步开销较大。

3.6 clang 编译器

该处使用 Omp_Parallel.c 作为代码样本，编写 clang 的 makefile 文件，文件名命名为“clang_makefile”，文件内容如下：

```
# Makefile for compiling Omp_Parallel.c with clang
    # Compiler
    CC = clang

    # Compiler flags
    CFLAGS = -fopenmp -O2 -Wall -Wextra -std=c11

    # Target executable
    TARGET = Omp_Parallel

    # Source files
    SRCS = Omp_Parallel.c

    # Object files
    OBJS = $(SRCS:.c=.o)

    # Default target
    all: $(TARGET)

    # Build target
    $(TARGET): $(OBJS)
        $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)

    # Compile source files into object files
    %.o: %.c
```

```
	$(CC) $(CFLAGS) -c $< -o $@

# Clean up generated files
clean:
	rm -f $(OBJS) $(TARGET)

# Phony targets
.PHONY: all clean
```

在终端输入命令“make -f clang_makefile”即可正常编译。

4 稀疏矩阵的并行算法

考虑矩阵计算时，稀疏矩阵是经常出现且不可忽视的，稀疏矩阵（Sparse Matrix）是指在一个矩阵中，大多数元素都是零的矩阵。由于这种矩阵的非零元素相对于总元素的数量非常少，在存储稀疏矩阵时，仅存储非零元素及其位置，可以显著减少内存使用。例如，使用 CSR（Compressed Sparse Row）或 CSC（Compressed Sparse Column）格式来存储稀疏矩阵比使用密集矩阵格式要节省很多空间。处理稀疏矩阵时，仅计算和处理非零元素，避免了对零元素的无用计算。这种方法提高了算法的计算效率，特别是在矩阵乘法、求解线性方程组等操作中。

因此对于两个稀疏矩阵相乘，我们可以采取特殊的处理办法，区别于正常的算法，如此就可以在大型工作中省去大量的时间。

4.1 CSR 格式

CSR（Compressed Sparse Row）格式是一种用于高效存储和操作稀疏矩阵的存储格式，广泛用于数值计算和线性代数中。CSR 格式通过压缩矩阵中的零元素来节省内存，并加速稀疏矩阵的计算操作。CSR 格式中主要用三类数据来表达：数值，列号，以及行偏移。CSR 不是三元组，而是整体的编码方式。数值和列号与 COO 一致，表示一个元素以及其列号，行偏移表示某一行的第一个元素在 values 里面的起始偏移位置。这样可以避免存储零元素，从而节省内存。在稀疏矩阵-稀疏矩阵乘法中，CSR 格式使得只需考虑非零元素进行计算，提高了效率。

4.2 基于 CSR 的稀疏矩阵相乘及并行算法

本文基于 CSR 存储稀疏矩阵，实现了矩阵串行相乘与并行相乘，自行设计了两个稀疏矩阵，其中每个矩阵具有 100000 个非零元素，原理与上文一致，但由于存储方式不同，代码设计也有了区别(如图 6)。使用一个临时数组 temp_values 来存储当前行的中间累加结果，遍历矩阵 A 当前行的非零元素，对于每个非零元素，查找矩阵 B 中对应行的非零元素，计算乘积并累加到 temp_values。最后将 temp_values 中的非零结果复制到结果矩阵 C 中。

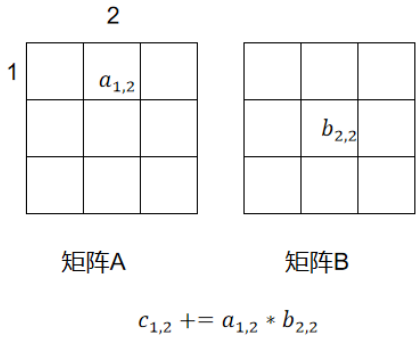


图6

核心代码如下：

```
 for (int j = A.row_ptr[i]; j < A.row_ptr[i + 1]; j++) {
    int a_col = A.col_ind[j];
    for (int k = B.row_ptr[a_col]; k < B.row_ptr[a_col + 1]; k++) {
        int b_col = B.col_ind[k];
        temp_values[b_col] += A.values[j] * B.values[k];
                }
            }
```

4.3 运行结果测试

本地环境配置为：ubuntu20.04，四核，内存大小 4G，四线程，记录不同规模下矩

阵非零元素个数此处固定为10^5个。

表5 稀疏矩阵乘法执行的时间（单位 s）

矩阵规模	1024*2048*1024	2048*4096*2048	4096*8192*4096	8192*16384*8192
串行	0.038931	0.031070	0.033135	0.073906
OpenMP	0.026879	0.011997	0.010372	0.016329

从表中可以看出使用 OpenMP 进行并行化显著减少了稀疏矩阵乘法的计算时间。对于较大的矩阵，OpenMP 的优势更加明显，能够提供更高的加速比。与前文对比，可以发现稀疏矩阵相乘的计算所需要的时间远远小于正常矩阵相乘的计算，故在对矩阵计算时，考虑到稀疏矩阵，并对稀疏矩阵进行特殊处理是十分必要的。

5 结束语

本文从矩阵相乘出发，系统地探讨了并行矩阵乘法的各种优化技术，包括利用 OpenMP 进行并行化、GCC 编译器优化选项的选择、MPI 在并行计算中的应用以及缓存优化策略。我们分析了不同优化手段在实际计算中的效果，并进行了详细的性能评估。通过对各种方法的实验和分析，我们发现不同的优化手段在不同的应用场景下能够显著提高矩阵乘法的效率。例如，OpenMP 在多核处理器上的并行化实现可以有效缩短计算时间，GCC 编译器的高级优化选项能够进一步提升代码的运行效率，而 MPI 则为分布式系统中的矩阵计算提供了高效的解决方案。除此之外，我们尝试了对稀疏矩阵的单独计算并取得了一个不错的效果，同时对并行算法还留有进一步优化的空间。

参考文献

[1] 苟悦宬. 使用 OpenMP+MPI 的矩阵乘法并行实现 [J]. 电脑与电信, 2022, (03): 77-80. DOI:10.15966/j.cnki.dnydx.2022.03.003.

[2] Catalán S ,Castelló A ,Igual D F , et al. Programming parallel dense matrix factorizations with look-ahead and OpenMP [J]. Cluster Computing, 2020, 23 (1): 359-375.

[3] 潘亮,郭改枝,宋鑫梦. 基于 OpenMP 矩阵相乘并行算法的设计 [J]. 宝鸡文理学院学报(自然科学版), 2014, 34 (01): 21-23. DOI:10.13467/j.cnki.jbuns.2014.01.008.

[4] 张艳华,刘祥港. 一种基于 MPI 与 OpenMP 的矩阵乘法并行算法 [J]. 计算机与现代化, 2011, (07): 84-87.

[5] 周灿. 基于 MPI 的矩阵运算并行算法研究[D]. 重庆大学, 2010.