第二次作业报告: Blocked CSR 数据结构与数据局部性优化

王晨宇 2100010654

数学科学学院

北京大学

2025 年 6 月 11 日

1 编译环境与程序运行指令

1.1 硬件与软件环境

本次实验在北京大学数学科学学院二号集群的计算节点上进行。编译器使用 GCC 4.8.5 (数院集群默认)，并启用 OpenMP 并行框架

1.2 程序版本与编译

除了原始的串行程序 Serial.cpp，本次作业提交文件中提交了另外两个并行计算版本的程序代码

- **serial.cpp**: 基础串行实现，使用标准 CSR 存储格式
- **parallel_row.cpp**: OpenMP 并行版本，按行分配计算任务
- **parallel_BCSR_sparse.cpp**: 基于 Blocked CSR 格式的 openmp 并行程序优化实现

编译时使用以下优化选项 (具体可见压缩包中的 Makefile 文件):

```
# 通用编译选项
CC = g++ -fopenmp -O3 -std=c++11
```

1.3 运行流程

本程序分别测试了 input1.csr 至 input5.csr 五个不同的稀疏矩阵上的 spmv 问题，同时对于并行程序分别测试了其线程数为 1，2，4，8，16 时的性能。程序运行分为以下步骤:

1. **编译程序**: 使用 `make all` 命令编译所有 C++ 程序。

```
make all
```

2. **运行程序**: 分别运行三个脚本以执行不同的程序，并将结果输出到当前文件夹。

   (a) 运行 serial.cpp：

```
./run.serial
```

   (b) 运行 parallel_row.cpp：

```
./run.parallel_row
```

   (c) 运行 parallel_BCSR_sparse.cpp：

```
./run.BCSR_sparse
```

3. **检查结果**: 在得到所有结果后，运行 python 程序 `check.py` 检查三个程序的输出结果是否一致。

```
python check.py output1 output2
```

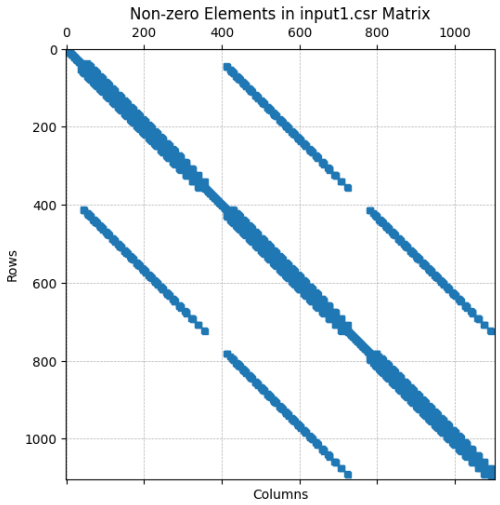
2 数据集特征与设计思路

2.1 稀疏矩阵数据集

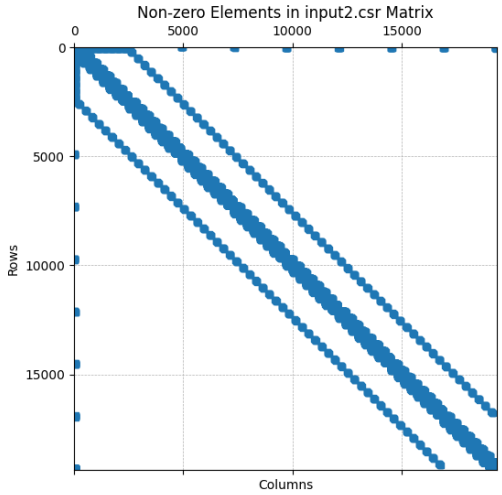
本次作业的五个测试稀疏矩阵的特征如下：

表 1: 测试系数矩阵特征

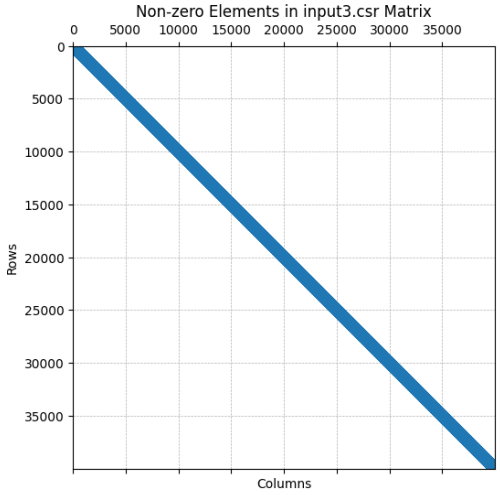
矩阵名称	维度	非零元数	密度 (%)	最大行宽
input1.csr	1,104	3,786	0.3106	7
input2.csr	19,362	83,443	0.0223	82
input3.csr	40,000	197,608	0.0124	5
input4.csr	40,000	197,608	0.0124	188
input5.csr	525,825	2,100,225	0.0008	4



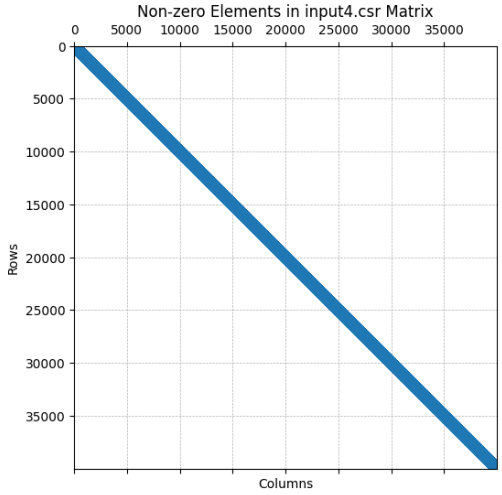
((a)) input1.csr



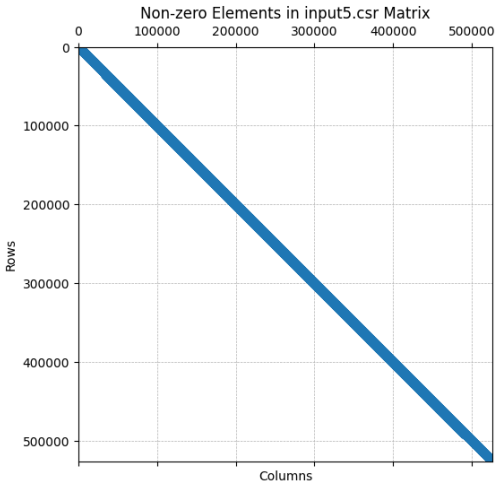
((b)) input2.csr



((c)) input3.csr



((d)) input4.csr



((e)) input5.csr

图 1: 稀疏矩阵非零元素分布图

2.2 数据分布观察

通过分析这些矩阵，我们发现以下重要特征：

- **局部聚集性**：五个矩阵均表现出不同程度的非零元聚集现象，如果单纯采用按行做并行计算的方式可能会频繁导致缓存未命中。这为 BCSR 格式提供了优化机会。
- **对角线集中**：五个矩阵中的非零元素均具有明显的对角线分布特征。
- **行宽分布均匀**：五个矩阵不同行的非零元元素数量分布较为均匀。这促使我们接下来并行计算算法中各个线程之间使用静态调度方式仍能取得不错的效果。

3 parallel_row.cpp 算法实现详解

Algorithm 1 parallel_row.cpp 并行算法

输入:
  A: CSR 格式的稀疏矩阵
  x_0: 初始向量
  $num\_iterations$: 迭代次数
输出:
  x: 解向量

procedure CALCULATING(A, x_0, $num\_iterations$)
  初始化 $x \leftarrow x_0$
  获取线程数 $num\_threads$
  计算块大小 $chunk \leftarrow \lceil A.num\_rows/num\_threads \rceil$
  for $iter = 1$ **到** $num\_iterations$ **do**
    并行区域开始
    for $i = 1$ **到** $A.num\_rows$ **do**
      $sum \leftarrow 0.0$
      for $k = A.row\_ptr[i]$ **到** $A.row\_ptr[i+1] - 1$ **do**
        $sum \leftarrow sum + A.values[k] \times x[A.col\_indices[k]]$
      end for
      $x[i] \leftarrow sum + x_0[i]$
    end for
    并行区域结束
  end for
  return x
end procedure=0

在这里面，我们采取了静态调度的方式，sum 变量为每个线程的私有变量，用于暂时存储求和的值，并最终写入公共变量 x 中。在编写程序的过程中，本次作业做了以下的优化努力：

- **静态调度策略**通过观察五个稀疏矩阵的特征，我们发现其行宽分布基本均匀，且每个线程的工作量与其处理的非零元个数几乎成正比。因此，我们采用了 OpenMP 的静态调度策略(`schedule(static, chunk)`)，将矩阵行均匀分配给各个线程。这种策略能够有效平衡线程间的工作负载，避免因任务分配不均导致的性能瓶颈。当然，这种做法严重依赖于我们对于稀疏矩阵的先验观察（行宽基本相等）。
- **减少 False Sharing 现象**为了减少多线程写入数据时的 False Sharing 现象，我们设计每个线程

处理连续的若干行数据。False Sharing 现象通常发生在多个线程同时修改位于同一缓存行的不同数据时，导致缓存行频繁失效，从而降低性能。通过让每个线程处理连续的行，我们确保每个线程写入的数据尽可能位于不同的缓存行中，从而减少缓存竞争，提升并行效率。

- **并行区域设计**我们将核心计算部分（3000 次迭代矩阵向量乘法和向量更新）放置在 OpenMP 并行区域内，并通过 `#pragma omp for` 指令将任务分配给多个线程，防止并行区多次创建引发性能下降。

在优化过程中，我们也尝试了对 x 向量进行 padding 的方法，以防止 false sharing 的发生。然而，实验结果表明，这种 padding 方法虽然在理论上可以减少线程间的缓存行竞争，但实际上却导致了内存访问命中率的显著下降，整体性能反而不如未采用 padding 时的表现。因此，最终并未采用 padding 处理方法，而是令每个线程负责写入数组的连续一块，以尽量避免 false sharing 现象。

4 Blocked CSR(parallel_BCSR_sparse.cpp) 算法实现详解

4.1 Blocked CSR 数据结构及转化

4.1.1 数据结构定义

在稀疏矩阵的存储中，Blocked Compressed Sparse Row（BCSR）格式是一种对传统 CSR 格式的扩展。它将矩阵划分为固定大小的块，以提高矩阵操作的效率。与此同时，我们发现即使将原矩阵分块，每个块内的元素依旧稀疏 (由 input1-input5 矩阵特征)，因此对于每个分块我们仍然采用 CSR 格式存储其数据。BCSR 格式的核心数据结构如下：

Listing 1: BCSR 数据结构

```
struct CSRBlock {
    int num_rows;
    int num_cols;
    int* row_ptr = nullptr;
    int* col_idx = nullptr;
    double* values = nullptr;
};

struct BCSRMatrix {
    int block_size;
    int num_block_rows;
    std::vector<int> block_row_ptr;
    std::vector<int> block_col_indices;
    std::vector<CSRBlock> blocks;
};
```

其中,`block_size` 表示每个块的大小,`num_block_rows` 表示矩阵划分为块后行的数量,`block_row_ptr` 类似于 CSR 格式中的 `row_ptr`，用于记录每个块行的起始位置，`block_col_indices` 类似于 CSR 格式中的 `col_indices`，用于记录每个块的列索引，`blocks` 则存储每个块的具体数据。每个块本身也是一个 CSR 格式的矩阵，因此对于每个分块的存储模式也是 CSR 格式。

4.1.2 BCSR 格式的并行转化

从传统的 CSR 格式转化为 BCSR 格式的过程可以概括为以下过程：

Algorithm 2 CSR 到 BCSR 并行转换

Require: CSR 矩阵 A，块大小 b

Ensure: BCSR 矩阵 B

0: 1. 计算块行数 $n_{blocks} \leftarrow \lceil A.rows/b \rceil$

0: 2. 初始化 B 的结构（行指针、列索引、块数据）

0: **并行阶段 1：统计非零块列**

每个线程处理若干块行，用原子操作统计全局非零块列数

0: **并行阶段 2：填充块结构**

每个线程独立处理分配的块行：

- 收集该块行的唯一块列索引

- 提取对应的 $b \times b$ 子矩阵存入 B(子矩阵的存储方式依旧以 CSR 格式)

0: 3. 返回构建完成的 BCSR 矩阵 B =0

如果读者对代码具体实现细节感兴趣，可查看 parallel_BCSR_sparse.cpp 文件中的函数: "csr_to_bcsr_parallel".

在构建好 BCSR 矩阵后，我们即可通过其进行 spmv 问题的 openmp 并行计算。

Algorithm 3 Blocked CSR 结构下的并行迭代算法

0: **并行区域:**

0: 将行划分为多个块，分配给每个线程

0: **对于** $iter = 1$ **到** $num\_iterations$:

0: 重置 $x\_next \leftarrow x\_0$ {并行拷贝}

0: **对于**每个块行 br: {并行块处理}

0: **对于**行 br 中的每个块:

0: 将块与 $x\_current$ 相乘，结果累加到 x_next

0: 交换 $x\_current$ 和 x_next {同步更新} =0

值得注意的是，在上述计算的每次循环中，我们一共开启过两次并行任务：

- 在每次迭代的开始，我们要将 x_next 向量初始化为 0。这里我们简单的采用静态调度即可获

得很好的效果。同时，我们合理选取 chunk_size 使得每个线程只写入连续的一块内存，这样可以在保证负载均衡的同时尽量避免 false sharing 现象的发生。

- 在每次计算的过程中，每一个计算单位为 Blocked CSR 格式中的一个 block. 由于位于同一行的 blocks 任务结果最终会写入同一段内存中，为了避免数据竞争，我们令每一行 Blocks 都分配给同一个线程，即每一个基本任务单元为一行 Blocks. 对于并行任务的调度方式，我们分别尝试了以下三种方法：1.static, 2.dynamic, 3. guided. 实验结果表明，尝试 static 调度方式的效果最好。这是因为本次作业提供的 5 个测试矩阵的行宽分布都很均匀，即使划分为 blocks 后每一行 blocks 的计算总量也相对一致。这一点观察恰好印证了我们的实验结果。因此，最终我们依旧选择采取 static 调度方式分配并行任务以实现尽可能好的性能。值得注意的是，此调度方法依赖于输入矩阵行间非零元素数量分布较为均匀的观察。

5 性能分析与对比

我们在五个测试矩阵上评估三个算法的性能：

5.1 整体性能对比

表 2: Serial.cpp 程序性能测试 (使用 CPU clock 计时吗，运行单次)

Matrix	input1.csr	input2.csr	input3.csr	input4.csr	input5.csr
Time (s)	0.030000	0.450000	0.760000	0.760000	10.040000

表 3: parallel_row.cpp 程序性能测试（运行 3 次取平均值）

Matrix	Threads = 1	Threads = 2	Threads = 4	Threads = 8	Threads = 16
input1	0.03311s (1.000×)	0.02863s (1.156×)	0.02325s (1.424×)	0.02344s (1.412×)	2.4249s (0.014×)
input2	0.34886s (1.000×)	0.26147s (1.334×)	0.16056s (2.173×)	0.11754s (2.973×)	10.2570s (0.034×)
input3	0.80496s (1.000×)	0.54237s (1.482×)	0.30123s (2.672×)	0.17518s (4.595×)	10.2990s (0.078×)
input4	0.80591s (1.000×)	0.45784s (1.760×)	0.30211s (2.668×)	0.18200s (4.428×)	10.2986s (0.078×)
input5	9.2703s (1.000×)	4.3263s (2.143×)	2.3001s (4.029×)	1.3061s (7.101×)	11.4712s (0.809×)

表 4: parallel_BCSR_sparse.cpp 程序性能测试（运行 3 次取平均值）

Matrix	Threads = 1	Threads = 2	Threads = 4	Threads = 8	Threads = 16
input1	0.07768s (1.000×)	0.05049s (1.538×)	0.04106s (1.892×)	0.05133s (1.516×)	0.76639s (0.101×)
input2	1.5882s (1.000×)	0.88097s (1.802×)	0.55911s (2.840×)	0.26430s (5.999×)	0.16102s (9.863×)
input3	2.0175s (1.000×)	1.0989s (1.835×)	0.66969s (3.012×)	0.34404s (5.863×)	4.8831s (0.413×)
input4	2.0028s (1.000×)	1.0383s (1.938×)	0.64171s (3.121×)	0.36162s (5.539×)	0.20407s (9.814×)
input5	37.725s (1.000×)	13.988s (2.700×)	7.3474s (5.136×)	3.9052s (9.660×)	2.9403s (12.829×)

5.2 结果与算法分析

通过分析同一算法在不同线程数、测试矩阵下的运行结果，我们发现：首先，从串行算法的表现来看，其运行时间随着矩阵规模的增大而线性增长，这与理论预期相符。

转向并行算法，对于行并行版本 (parallel_row.cpp)，其在线程数数量不多的情况下（线程数小于等于 8）时，程序运行耗时随着线程数的增加显著减少。特别的，对于大规模矩阵（如 input5.csr），而对于小规模矩阵（input1.csr），其线程的调度开销与内存竞争相比于其原本计算量较大，因此其性能并不显著优于串行版本。值得注意的是，当线程数为 16 的时候，行并行版本在每一个测试矩阵上的性能均明显差于串行程序。原因具体有两点：

- Openmp 各线程共享同一块内存区域，线程数过多会导致每个线程所分配到的内存变少，进而会导致频繁的 cache miss。（主要原因）
- 随着线程数的增加，并行算法中的任务调度开销增加，进而影响程序性能。（次要原因）

相比之下，BCSR 并行算法在处理大规模稀疏矩阵时展现出了明显的加速比优势。在 input5 上，即使使用 16 线程，BCSR 版本仍能保持高效的加速比，这得益于块压缩存储格式对内存访问模式的优化。块存储不仅提高了数据的局部性，还增加了计算密度，从而缓解了内存带宽的压力。然而，由于 BCSR 版本由于额外的预处理开销，这也使得 BCSR 版本的算法更适用于使用此稀疏矩阵做多次向量乘问题迭代。值得注意的是，BCSR 版本的并行程序除了在 input1.csr, input3.csr 两个测试矩阵的 16 线程版本程序中遭遇了与行并行版本 (parallel_row.cpp) 相同的性能瓶颈问题，其在其余测试矩阵上的加速比仍随着线程数的增加而增加。这是因为 BCSR 并行算法充分利用了数据局部性，使得其对内存读取更加友好。这也使得 BCSR 并行算法相比于行并行算法可能有着更高的性能上限。然而，由于数院集群限制最多使用 16 个线程，我们暂时无法测试其在更多线程下的程序性能。

值得注意的是，对于大型稀疏矩阵 input5.csr，两个并行计算的算法在线程数等于 2 和 4 的时候，取得的加速比均分别大于 2 和 4。这主要归功于多线程在内存数据读写方面的优势。具体来说，稀疏矩阵的存储格式（如 CSR）通常包含非零元素及其索引信息，多线程可以并行访问不同的非零元素块，从而减少内存访问的竞争和延迟。同时，每个线程充分利用局部缓存，减少 Cache Miss 的频率，进一步提升数据读取效率，使得最终加速比大于线程数。

最后，我们对比一下两个并行算法 (行并行算法和 BCSR 算法) 的程序性能。我们发现，虽然 BCSR 算法在理论上具有更好的数据局部性，但在相同的测试矩阵与线程数下，行并行算法的程序耗时小于 BCSR 算法。这是因为 BCSR 矩阵引入了更加复杂的存储结构与更复杂的调度问题，导致线程的调度开销的增加超出了数据局部性的优化。然而，由上述讨论，不同于行并行算法，由于 BCSR 算法内存读写友好的特点，其加速比依然在随着线程数的增加而增加，因此其可能会更适合多线程并行任务。

5.3 局限性与未来优化方向

由前文的分析与总结，我们的 SpMV 算法未来还有如下的优化方向:

1. 在执行并行任务的阶段，由于我们观察到本次作业的五个测试矩阵均呈现出行宽相同的特性，我们选取静态调度的策略分配任务。未来我们可以将我们的算法拓展到更一般的稀疏矩阵上。比如可以事先通过分析系数矩阵的非零元素分配特点，预先将每个线程分配给不同的行，从而在迭代中维持调度开销不大的同时使其负载更加均衡。

2. 在我们的 BCSR 算法中，我们采用了固定大小的分块策略（block size = 50）. 未来我们可以对其加以改进，对于非零元素密集的区域采取较小的分块，而对稀疏的区域采取更大的分块，从而进一步维护负载均衡。

6 程序正确性检验 check.py

```
-bash-4.2$ python check.py output1_serial output1_parallel_BCSR_sparse_threads16
\Comparing File 1: 'output1_serial' (Reference)
With File 2     : 'output1_parallel_BCSR_sparse_threads16' (Test)
Tolerances      : rtol=1.0e-05, atol=1.0e-06

--- Reference Vector (File 1) Stats ---
Reference Vector Norms:
  L1 (Sum of abs values): 5.125842e+03
  L2 (Euclidean)        : 1.850188e+02
  L-inf (Max abs value) : 1.230253e+01
--- Vector Under Test (File 2) Stats ---
Test Vector Norms:
  L1 (Sum of abs values): 5.125842e+03
  L2 (Euclidean)        : 1.850188e+02
  L-inf (Max abs value) : 1.230253e+01
--- Error Analysis ---
Difference Vector (Ref - Test) Stats:
Difference Vector Norms:
  L1 (Sum of abs values): 2.995104e-13
  L2 (Euclidean)        : 3.128531e-14
  L-inf (Max abs value) : 1.065814e-14

Vector length: 1104
Maximum Absolute Error (L-inf norm of difference): 1.065814e-14
Maximum Relative Error (where ref > 1.0e-15): 6.049534e-15
--- Closeness Check ---
SUCCESS: Vectors are close (rtol=1.0e-05, atol=1.0e-06).
```

((a)) input1.csr

```
-bash-4.2$ python check.py output2_serial output2_parallel_BCSR_sparse_threads16
Comparing File 1: 'output2_serial' (Reference)
With File 2     : 'output2_parallel_BCSR_sparse_threads16' (Test)
Tolerances      : rtol=1.0e-05, atol=1.0e-06

--- Reference Vector (File 1) Stats ---
Reference Vector Norms:
  L1 (Sum of abs values): 1.114683e+05
  L2 (Euclidean)        : 9.230720e+02
  L-inf (Max abs value) : 1.128389e+01
--- Vector Under Test (File 2) Stats ---
Test Vector Norms:
  L1 (Sum of abs values): 1.114683e+05
  L2 (Euclidean)        : 9.230720e+02
  L-inf (Max abs value) : 1.128389e+01
--- Error Analysis ---
Difference Vector (Ref - Test) Stats:
Difference Vector Norms:
  L1 (Sum of abs values): 8.162971e-12
  L2 (Euclidean)        : 1.851075e-13
  L-inf (Max abs value) : 1.065814e-14

Vector length: 19362
Maximum Absolute Error (L-inf norm of difference): 1.065814e-14
Maximum Relative Error (where ref > 1.0e-15): 1.064647e-15
--- Closeness Check ---
SUCCESS: Vectors are close (rtol=1.0e-05, atol=1.0e-06).
```

((b)) input2.csr

```
-bash-4.2$ python check.py output3_serial output3_parallel_BCSR_sparse_threads16
Comparing File 1: 'output3_serial' (Reference)
With File 2     : 'output3_parallel_BCSR_sparse_threads16' (Test)
Tolerances      : rtol=1.0e-05, atol=1.0e-06

--- Reference Vector (File 1) Stats ---
Reference Vector Norms:
  L1 (Sum of abs values): 1.952156e+05
  L2 (Euclidean)        : 1.140720e+03
  L-inf (Max abs value) : 1.088345e+01
--- Vector Under Test (File 2) Stats ---
Test Vector Norms:
  L1 (Sum of abs values): 1.952156e+05
  L2 (Euclidean)        : 1.140720e+03
  L-inf (Max abs value) : 1.088345e+01
--- Error Analysis ---
Difference Vector (Ref - Test) Stats:
Difference Vector Norms:
  L1 (Sum of abs values): 2.325738e-11
  L2 (Euclidean)        : 2.423769e-13
  L-inf (Max abs value) : 1.065814e-14

Vector length: 40000
Maximum Absolute Error (L-inf norm of difference): 1.065814e-14
Maximum Relative Error (where ref > 1.0e-15): 2.031866e-15
--- Closeness Check ---
SUCCESS: Vectors are close (rtol=1.0e-05, atol=1.0e-06).
```

((c)) input3.csr

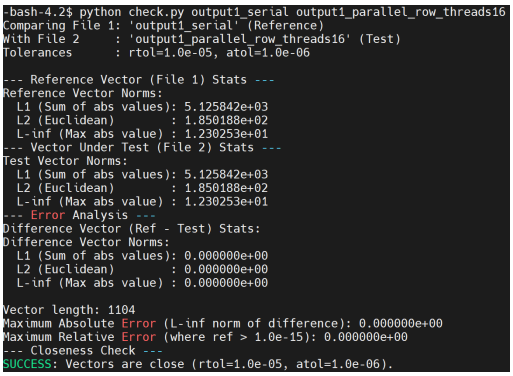
```
-bash-4.2$ python check.py output4_serial output4_parallel_BCSR_sparse_threads16
Comparing File 1: 'output4_serial' (Reference)
With File 2     : 'output4_parallel_BCSR_sparse_threads16' (Test)
Tolerances      : rtol=1.0e-05, atol=1.0e-06

--- Reference Vector (File 1) Stats ---
Reference Vector Norms:
  L1 (Sum of abs values): 1.863688e+05
  L2 (Euclidean)        : 1.097992e+03
  L-inf (Max abs value) : 9.714672e+00
--- Vector Under Test (File 2) Stats ---
Test Vector Norms:
  L1 (Sum of abs values): 1.863688e+05
  L2 (Euclidean)        : 1.097992e+03
  L-inf (Max abs value) : 9.714672e+00
--- Error Analysis ---
Difference Vector (Ref - Test) Stats:
Difference Vector Norms:
  L1 (Sum of abs values): 2.838459e-11
  L2 (Euclidean)        : 1.990861e-13
  L-inf (Max abs value) : 3.552714e-15

Vector length: 40000
Maximum Absolute Error (L-inf norm of difference): 3.552714e-15
Maximum Relative Error (where ref > 1.0e-15): 1.063332e-15
--- Closeness Check ---
SUCCESS: Vectors are close (rtol=1.0e-05, atol=1.0e-06).
```

((d)) input4.csr

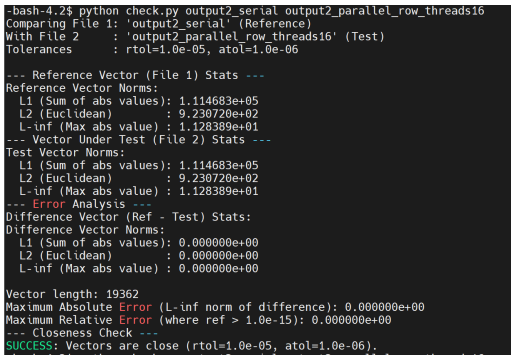
```
-bash-4.2$ python check.py output5_serial output5_parallel_BCSR_sparse_threads16
Comparing File 1: 'output5_serial' (Reference)
With File 2     : 'output5_parallel_BCSR_sparse_threads16' (Test)
Tolerances      : rtol=1.0e-05, atol=1.0e-06

--- Reference Vector (File 1) Stats ---
Reference Vector Norms:
  L1 (Sum of abs values): 2.366222e+06
  L2 (Euclidean)        : 3.609412e+03
  L-inf (Max abs value) : 8.952467e+00
--- Vector Under Test (File 2) Stats ---
Test Vector Norms:
  L1 (Sum of abs values): 2.366222e+06
  L2 (Euclidean)        : 3.609412e+03
  L-inf (Max abs value) : 8.952467e+00
--- Error Analysis ---
Difference Vector (Ref - Test) Stats:
Difference Vector Norms:
  L1 (Sum of abs values): 1.432833e-10
  L2 (Euclidean)        : 4.111690e-13
  L-inf (Max abs value) : 3.552714e-15

Vector length: 525825
Maximum Absolute Error (L-inf norm of difference): 3.552714e-15
Maximum Relative Error (where ref > 1.0e-15): 1.109351e-15
--- Closeness Check ---
SUCCESS: Vectors are close (rtol=1.0e-05, atol=1.0e-06).
```

((e)) input5.csr

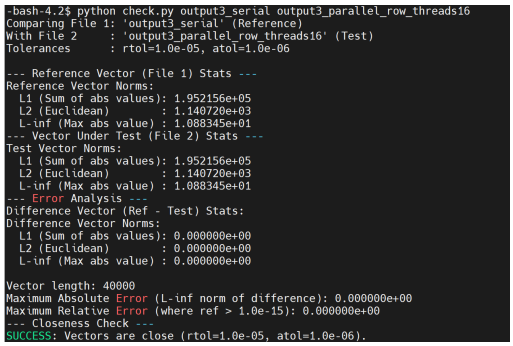
图 2: 程序正确性检验 Serial vs BCSR 16 threads



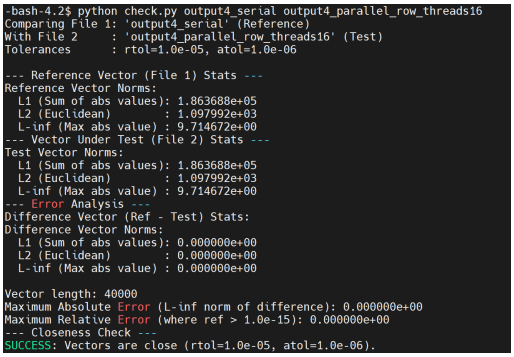
((a)) input1.csr



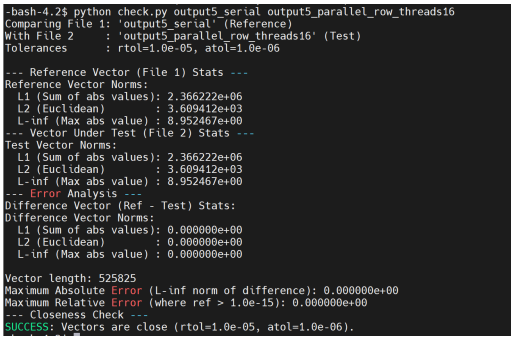
((b)) input2.csr



((c)) input3.csr



((d)) input4.csr



((e)) input5.csr

图 3: 程序正确性检验 Serial vs Parallel Row 16 threads