并行计算 实验二

PB19071501 李平治

I. 问题描述

输入一个稀疏矩阵（非零元比例<1%）、一个向量，求两者的乘积，使用MPI实现。

$$Ax = b$$

(A为稀疏矩阵)

II. 算法设计

i. 算法策略

本次实验采用的算法为Row Partition，测试用例为均匀分布产生的稀疏矩阵。

具体来说，记矩阵A行数为n，进程数为`n_proc`，则该计算任务可以分为
$C = \{b_1 = A_1 \cdot x, b_2 = A_2 \cdot x, \dots, b_n = A_n \cdot x\}$。将$C$均匀分给这`n_proc`个进程。

为了节省空间，矩阵生成、读取和运算过程均采用CSR (Compressed Sparse Row) Format。

以下为PCAM方法学描述：

- 划分：将稀疏矩阵按每行划分
- 通信：结构化通信，由0号进程进行矩阵读取和数据分发
- 组合：将每个进程所计算的几行组合成一个子矩阵，进行和`rhs`的计算
- 映射：由于该算法具有结构化的局部通信，映射设计显然

ii. 数据结构与关键代码

本实验测试用例矩阵采用$n \times n$方阵，非零元素位置由均匀分布产生，非零元素大小也为均匀分布产生。

- MPI消息数据结构

```
typedef struct InfoT {
    int num_rows;
    int num_elem;
    int elem_displs;
} InfoT;
```

其中num_rows记录该进程所计算的行数，num_elem记录该进程所计算行的非零元素数，elem_displs记录该进程所计算的　非零元素起始位置。

- CSR矩阵

```
class CSRMatrix{
public:
    int n;
    int nnz;
    std::vector<int> csr_row_ptr;
    std::vector<int> csr_col_idx;
    std::vector<double> csr_val;
    CSRMatrix(){
        csr_row_ptr.clear();
        csr_col_idx.clear();
        csr_val.clear();
    }
    ~CSRMatrix(){
        csr_row_ptr.clear();
        csr_col_idx.clear();
        csr_val.clear();
    }
    void read_csr_matrix(int n_in, int nnz_in, string filepath);
};
```

其中n为方阵边长，nnz为非零元素数量，csr_row_ptr csr_col_idx csr_val分别为CSR格式的非零元素行指针、列索引和值大小。

- 计算各进程所计算行信息

```
for(int i=0; i<n_proc; i++) {
    info_procs[i].num_rows = num_rows;
    info_procs[i].num_elem = MM.csr_row_ptr[i * num_row_per_proc + num_rows_procs[i]] - MM.csr_row_ptr[i * num_row_per_proc];
    num_elems_procs[i] = info_procs[i].num_elem;
    info_procs[i].elem_displs = MM.csr_row_ptr[i * num_row_per_proc];
    elem_displs_procs[i] = info_procs[i].elem_displs;
}
```

```
proc_row_ptr = std::vector<int>(num_row_per_proc+1).data();
MPI_Scatterv(MM.csr_row_ptr.data(), num_rows_procs, row_displs_procs, MPI_INT,
proc_row_ptr, num_row_per_proc, MPI_INT, 0, MPI_COMM_WORLD);
proc_row_ptr[num_row_per_proc] = num_elem + proc_row_ptr[0];

proc_col_idx = std::vector<int>(num_elem).data();
MPI_Scatterv(MM.csr_col_idx.data(), num_elems_procs, elem_displs_procs, MPI_INT,
proc_col_idx, num_elem, MPI_INT, 0, MPI_COMM_WORLD);

proc_val = std::vector<double>(num_elem).data();
MPI_Scatterv(MM.csr_val.data(), num_elems_procs, elem_displs_procs, MPI_DOUBLE,
proc_val, num_elem, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

III. 实验评测

i.实验配置

本次实验在服务器上运行，相关配置如下:

- **24v**CPUs (Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz)
- MemTotal: 188GB
- Linux version 3.10.0-1160.el7.x86_64
- **g++** (conda-forge gcc 11.2.0-16) 11.2.0
- mpirun (Open MPI) 4.0.2

矩阵生成参数:

- 非零元占比 NON_ZERO_SCALE = 0.007
- 方阵行数 MATRIX_WIDTH = 1000000

ii.实验结果

a. 正确性验证

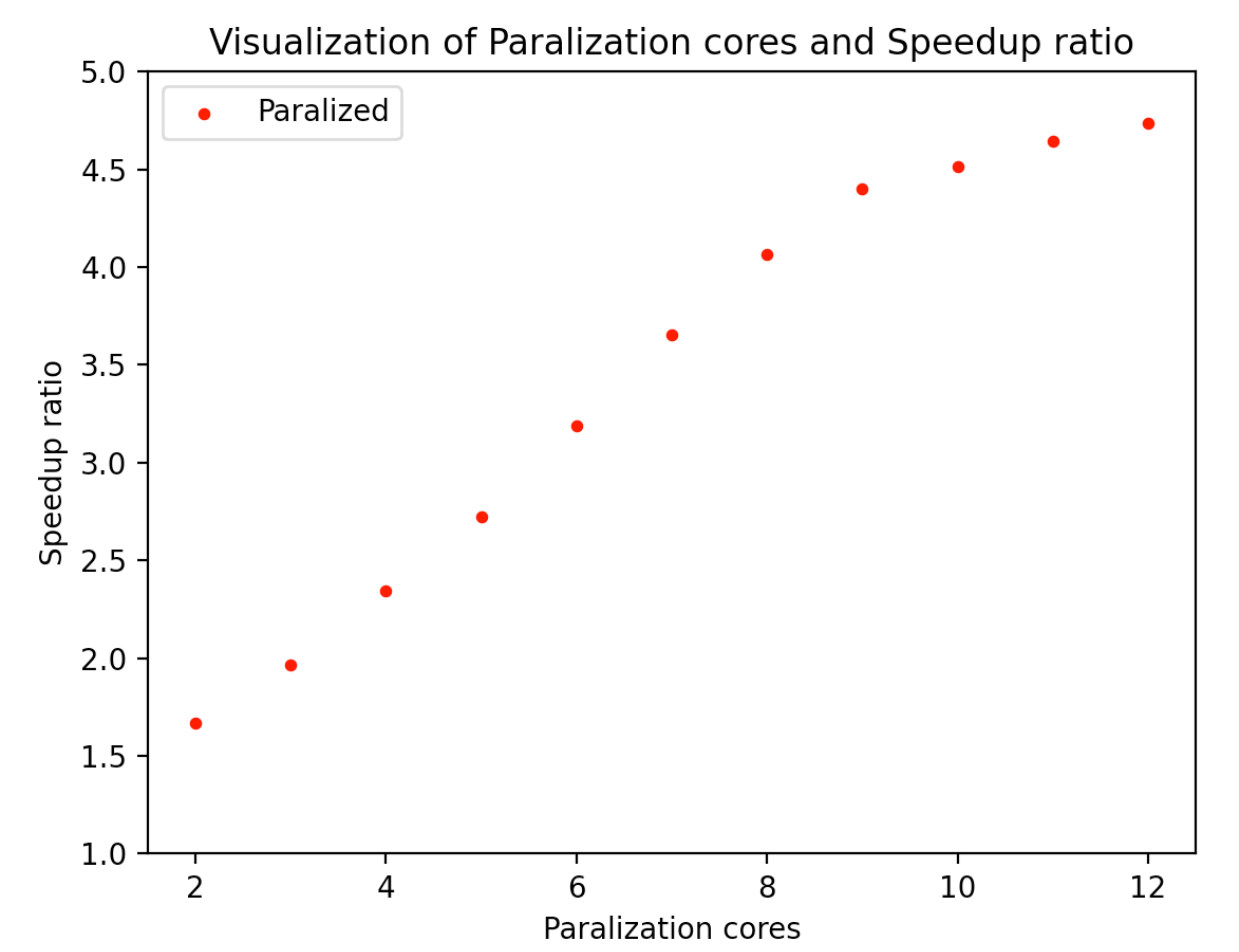
将串行与并行结果及实际结果用Python脚本对比，发现相同。

b.加速比分析

选择处理器核数2-12的并行算法和串行算法进行分析:

并行处理器核数	程序运行时间/ms	加速比
串行	89.593	None
2	53.709	1.670
3	45.551	1.969
4	38.252	2.345
5	32.934	2.723
6	28.093	3.193
7	24.509	3.660
8	22.032	4.071
9	20.371	4.403
10	19.838	4.521
11	19.293	4.649
12	18.920	4.741

加速比和并行核数的可视化:



可以看出，尽管没有取得线性加速比，但在并行数不高的情况下获得了非常好的结果。

在$N > 9$以后，由于通信等操作限制，并行效果边界收益开始快速减小；因此在可能的情况下，$N = 9$为最佳选择。

IV. 结论

本次实验考量了不同并行数下的并行算法效果，结果证明并行算法存在边界收益衰弱的表现。

参考

[1] Fan et al. A Study of SpMV Implementation using MPI and OpenMP on Intel Many-Core Architecture.

附录

spmv_serial.cpp为串行程序代码，spmv_parallel.cpp为并行程序代码，sparse_matrix_generator.py为系数矩阵CSR格式生成程序。