COL380 A4

BlockCSR Multiplication

Ansh Agrawal 2022CS51135

Contents

1 Overview of Key Functions

In this project, several functions are utilized to implement the parallel sparse matrix multiplication. The following is a brief one-line explanation for each key function used in the implementation:

- `read_size_file`: Reads the input folder's `size` file to obtain the total number of matrices and the block size, and computes the matrix distribution for each MPI process. Each rank find the start and end of the segment it has to
- `read_file`: Loads an individual matrix from a file in the block-compressed sparse row (BCSR) format and initializes the corresponding data structures.
- `send_bcsr`: Sends a BCSR matrix from one MPI process to another, ensuring that the matrix data (dimensions, row pointers, column indices, and values) is transferred correctly.
- `recv_bcsr`: Receives a BCSR matrix on an MPI process, reconstructing the matrix structure using the received data.
- `CSR_mul`: Performs the multiplication of two matrices in BCSR format; this function employs a CUDA kernel for block-level multiplication and leverages OpenMP for parallelism on the CPU.
- `bcsr_mul_kernel`: A CUDA kernel that computes the product of two sparse matrices by multiplying non-zero blocks and accumulating results using atomic operations.

2 Algorithm for Matrix Multiplication

The matrix multiplication algorithm in this project is designed to efficiently compute the product of a sequence of sparse matrices in block-compressed sparse row (BCSR) format. The algorithm leverages distributed, shared-memory, and GPU parallelism. The following steps summarize the high-level idea:

1. **Input Distribution:**
   - Each MPI process is assigned a subset of the input matrices from the folder. This is determined by reading the input file `size` via `read_size_file`. The ranks equally divide the task within themselves.
2. **Local Matrix Reading:**
   - Each MPI process concurrently loads its assigned matrices using the `read_file` function, which parses the BCSR formatted data.
3. **Pairwise Multiplication (Local Reduction):**
   - Each MPI process performs local pairwise multiplication of matrices. When multiplying two matrices, the `CSR_mul` function is invoked.

   - The function `CSR_mul` employs both OpenMP for parallelizing operations over CPU cores and a CUDA kernel (`bcsr_mul_kernel`) for accelerating block-level multiplication on the GPU.
   - Multiplication is performed iteratively, reducing the number of matrices by roughly half at each iteration until a single local result is obtained.

4. **Global MPI Reduction:**
   - Partial results from local multiplications are distributed across MPI processes.
   - Through a reduction tree structure, processes pair up and exchange results using `send_bcsr` and `recv_bcsr`.
   - A receiving process multiplies its local result with the received matrix, and this process continues until the final global product is accumulated at MPI rank 0.

5. **Output Generation:**
   - The final result, residing on the root process (MPI rank 0), is written to an output file `matrix` in the same BCSR format using `write_final_ans`.

This hierarchical approach—employing MPI for inter-node communication, OpenMP for intra-node CPU parallelism, and CUDA for GPU acceleration—ensures that the multiplication is performed efficiently even when dealing with large and sparse matrices.

3 Optimisation with Data Storage and GPU Computation

This section describes the storage format for the sparse matrices and explains how the chosen format and CUDA kernel implementation contribute to efficient computation.

3.1 Block Compressed Sparse Row (BCSR) Format

The matrices in this project are stored in the Block Compressed Sparse Row (BCSR) format. In this format, non-zero values are stored block-wise, which is particularly advantageous when dealing with sparsity at the block level. The benefits of using BCSR include:

- **Memory Efficiency:** Only non-zero blocks of the matrix are stored, which greatly reduces memory usage for sparse matrices.
- **Improved Access Patterns:** By grouping non-zeros into blocks of size $k \times k$, memory accesses are more coalesced and better suited to the memory hierarchy of GPUs.
- **Simplified Computation:** Block-level operations allow the kernel to process multiple values simultaneously, leading to better utilization of parallel execution units.

The BCSR format is implemented as follows:

```
struct BCSR {
    int rows, cols;                // Matrix dimensions
    int nz;                        // Number of non-zero blocks
    int* rowptr;                   // Row pointers indicating
       block start positions
    int* colind;                   // Column indices for each
       non-zero block
    unsigned long long* values;    // Data for non-zero blocks (
       flattened k*k arrays)
};
```

This structure represents a matrix by recording its dimensions, the number of non-zero blocks, and the data associated with those blocks. The `rowptr` array delineates the start of each row in the block-level view, while the `colind` and `values` arrays store the column locations and the actual block data respectively.

3.2 CUDA Kernel for Sparse Matrix Multiplication

The CUDA kernel, `bcsr_mul_kernel`, is designed to multiply two matrices stored in BCSR format. Its design takes advantage of shared memory and atomic operations to efficiently compute the dot-product of non-zero blocks. Key aspects of the kernel include:

- **Thread Configuration:** Each grid block corresponds to a row of blocks in matrix A, and each thread within the block handles the computation for an element in the resulting block (i.e., thread index `tid` determines its row and column within the block).

- **Shared Memory Usage:** Two shared memory arrays are declared to hold the block data from matrices A and B. This reduces the number of global memory accesses and speeds up computation.

```
extern __shared__ unsigned long long shmem[];
unsigned long long* Ash = shmem;
unsigned long long* Bsh = shmem + k*k;
```

- **Block Multiplication:** The kernel iterates over the non-zero blocks in the current row of matrix A and, for each block, fetches the corresponding block from matrix B. Then, it computes the dot-product for each element in the resulting block:

```
unsigned long long sum = 0;
for (int kk = 0; kk < k; ++kk) {
    sum += Ash[bi*k + kk] * Bsh[kk*k + bj];
}
```

- **Atomic Accumulation:** Since multiple threads might update the same position in the output matrix, the kernel uses an atomic operation to safely accumulate the results:

```
atomicAdd(&C_vals[pos * k*k + bi*k + bj], sum);
```

- **Synchronization:** The use of `__syncthreads()` ensures that all threads have completed loading their respective block data before proceeding with the multiplication, preserving data consistency.

In summary, the combination of the BCSR format and the specialized CUDA kernel significantly enhances performance by reducing memory overhead, providing coalesced memory accesses, and effectively leveraging the parallel processing capabilities of the GPU. This design enables the multiplication of large, sparse matrices in a highly efficient and scalable manner.

3.3 MPI Reduction for Global Matrix Multiplication

After performing local multiplications on individual MPI processes, the final step is to reduce all partial results into a single global matrix. The reduction is implemented using a hierarchical strategy in which processes pair up, exchange their computed matrices, and perform additional multiplications. The following code snippet outlines this approach:

```
int itersize = 1;

while (itersize < size) {
    if (rank % (itersize * 2)) {
        // Send the computed result to the designated partner.
        int dest = rank - itersize;
        send_bcsr(dest, final_result, k);
        break;
    } else {
        if (rank + itersize < size) {
            // Receive a partial result from the partner process.
            int src = rank + itersize;
            recv_bcsr(src, input, k);
            BCSR C;
            // Multiply the received matrix with the current
               result.
            CSR_mul(final_result, input, C, k);
            // Update final_result with the new multiplication
               result.
            final_result = C;
        }
    }
    itersize *= 2;
}
```

This reduction strategy allows for a balanced tree-like aggregation of results, minimizing the overall communication overhead while ensuring that each MPI process contributes to the final multiplication result. The use of pairwise communication and reduction helps hide the latency associated with MPI transfers by overlapping communication with computation.

4 Performance Analysis

The performance of our sparse matrix multiplication was evaluated by running experiments with different numbers of matrices (10, 50, 100, 150, and 200) across varying node configurations. The following timing results (reported in `real` time) were observed:

4.1 Experimental Results

- **2 Nodes:**
  - 10 matrices: 6.134 s
  - 50 matrices: 7.682 s
  - 100 matrices: 9.865 s
  - 150 matrices: 13.304 s
  - 200 matrices: 15.461 s
- **6 Nodes:**
  - 10 matrices: 11.797 s
  - 50 matrices: 12.465 s
  - 100 matrices: 12.490 s
  - 150 matrices: 13.288 s
  - 200 matrices: 14.458 s
- **4 Nodes:**
  - 10 matrices: 6.041 s
  - 50 matrices: 5.183 s
  - 100 matrices: 5.463 s
  - 150 matrices: 6.130 s
  - 200 matrices: 7.474 s

4.2 Analysis of Trends

The results reveal several interesting trends regarding how the number of nodes influences overall execution time:

- **Impact of Problem Size:** As the number of matrices increases, the execution time increases for all node configurations. This is expected because more matrix multiplications and communications are required. The scaling behavior is more pronounced for larger inputs (150 and 200 matrices).
- **Effect of Node Count:**
  - With **2 Nodes**, the execution time steadily increases with the number of matrices. The workload per node is higher, but communication overhead is relatively low.

  - With **6 Nodes**, the absolute times are higher for the smaller workloads (e.g., 11.797 s for 10 matrices) compared to the 2-node case. This indicates that when the problem size is small, the overhead of managing more MPI nodes and the additional inter-node communication can negate the benefits of parallel processing. As the workload increases, the difference between 6 nodes and fewer nodes tends to narrow because the computation time dominates the fixed communication costs.
  - With **4 Nodes**, we observe the best performance overall. For instance, with 50 matrices the time is only 5.183 s, which is notably lower than the corresponding times for both 2 and 6 nodes. This suggests that there exists an optimum balance between reducing the workload per node and minimizing inter-node communication overhead. In our experiments, 4 nodes provided that balance, leading to improved scaling.

- **Communication Overhead vs. Computation:** The hierarchical reduction using MPI introduces communication overhead that becomes more significant when the workload is light (i.e., fewer matrices). With more nodes, although the computation is distributed further, the communication and synchronization overhead can increase the overall time. As the workload per node increases (more matrices), the benefit of parallelizing the computation over more nodes may eventually pay off, but only when the communication costs are amortized by the heavier computation.

In summary, the performance analysis indicates that while distributing the workload across nodes can reduce computation time when the matrix count is high, excessive distribution (as in the 6-node case for smaller inputs) introduces communication overhead that can degrade performance. The 4-node configuration in this experiment strikes an effective balance between parallel computation and communication, yielding the lowest overall execution times.

4.3 Performance on Medium Test Case

For the medium test case, using 2 nodes resulted in a real execution time of 7.021 seconds, which decreased to 6.280 seconds with 4 nodes. Notably, further scaling to 6 nodes significantly improved performance, reducing the execution time to 3.846 seconds, indicating that for this test case, a higher node count effectively balances the workload and communication overhead.