Programming Methods in Scientific Computing

Winter semester 2019/2020
Prof. Dr. Marc Alexander Schweitzer
Clelia Albrecht and Albert Ziegenhagel

Exercise sheet 5.

The assignment corresponding to this exercise sheet can be found at `https://classroom.github.com/g/ajtW5mB1`. Accept the assignment and download the starter code repository. Copy your code from exercise sheet 4 to the corresponding files in this repository.

Parallelization

Exercise 1. (Install an MPI implementation)

We first need to make sure that an implementation of the MPI standard is installed on your computer. There are multiple vendors that provide an MPI implementation. We will use `OpenMPI` on Linux and macOS and Microsoft MPI (`MSMPI`) on Windows.

a) Install MPI on your computer:

- **Linux:** Open a terminal and run

  ```
  sudo apt-get update
  sudo apt-get install libopenmpi-dev
  ```

  for Ubuntu. For other Linux distributions, choose your corresponding package manager.
- **Windows:** You can download Microsoft MPI from `https://www.microsoft.com/en-us/download/details.aspx?id=100593`. You will have to download both packages `msmpisetup.exe` and `msmpisdk.msi`. After downloading the installers, install them by executing the installers and following the instructions.
- **macOS:** Open a terminal and run

  ```
  brew install open-mpi
  ```

  to install OpenMPI.

Exercise 2. (A Partition class)

As a first step for parallelizing our linear algebra data structures and operations, we write a data structure that is able to handle the partition of our linearly indexed data to several processes $0 \leq p \leq P$. While our data is indexed by a global index i_{global} in the range $[0, N)$, every process p just has access to a part of the data, indexed by a local index i_{local} in the range $[r_p, r_{p+1}) \subset [0, N)$. Here r_p is the global index which marks the start of the data owned by process p and $r_{p+1} - r_p = N_p$ is its size, with $r_0 = 0$ and $r_{P+1} = N$. Note that, as the r_p are ordered, we have $N_p \geq 0$ for all processes p. The indices $\{r_p\}$ are called the partition of our data.

Write a `ContiguousParallelPartition` class that helps you to handle the partition of your data across several processes. This class should provide a default constructor

```
explicit ContiguousParallelPartition();
```

and a constructor that is able to take an MPI communicator and a pre-existing partition list

```
explicit ContiguousParallelPartition(MPI_Comm communicator,
                                     std::vector<int> partition);
```

Furthermore, we want getter functions for the MPI communicator, the size of the local data N_p for our process p as well as the size of the local data of an arbitrary process q, N_q:

```
MPI_Comm communicator() const;
int local_size() const;
int local_size(int process) const;
```

Additionally, write a getter function for the global data size N:

```
int global_size() const;
```

An important functionality of your class should be to match global indices to processes. To this end, write the following functions:

- A function that finds the owner process to a global index:

  ```
  int owner_process(int global_index) const;
  ```

- A function that determines whether a global index belongs to your local process:

  ```
  bool is_owned_by_local_process(int global_index) const;
  ```

- A function that determines whether a global index belongs to an arbitrary process:

  ```
  bool is_owned_by_process(int global_index, int process) const;
  ```

Furthermore, you will need to switch between local and global indices using the following functions:

```
int to_global_index(int local_index) const;
int to_local_index(int global_index) const;
```

Last, please write two free functions that take care of the actual creation of the partition. The first one,

```
ContiguousParallelPartition create_partition(MPI_Comm communicator,
                                             int local_size);
```

should create a parallel partition by taking an MPI communicator and the local data size for every process.
The second one creates a uniform partition (the local data has the same size on every process) taking just the global data size N:

```
ContiguousParallelPartition create_uniform_partition(MPI_Comm communicator,
                                                     int global_size);
```

Exercise 3. (A parallelized vector class)

a) Parallelize the `Vector` class of your linear algebra library. To this end, expand the interface by adding the following functions:

   - A constructor that takes a partition

     ```
     explicit Vector(ContiguousParallelPartition partition);
     ```

   - A constructor that takes an MPI communicator and the local data size

     ```
     explicit Vector(MPI_Comm communicator, int local_size);
     ```

   - A constructor that takes an MPI communicator and the local data for this process

     ```
     explicit Vector(MPI_Comm communicator,
                     std::initializer_list<value_type> init);
     ```

   - A getter function for a `ContiguousParallelPartition`

     ```
     const ContiguousParallelPartition& partition() const;
     ```

   What do you have to change regarding the implementation? Make sure all the sequential tests still work. All constructors that are not explicitly parallelized should take `MPI_COMM_SELF` as MPI communicator.

b) Parallelize the vector operations (`equals`, `assign`, `add`, `subtract`, `multiply` and `dot_product`). Note: this step does not require the parallelization of the matrix-vector product. Which parts of your code do you actually have to change? Make sure all sequential tests for your operations still work.

c) Test the sequential and parallel performance of the vector addition and the inner product of two vectors. Measure the time of both operations on 1, 2 and 4 processors (and 8, if your hardware provides as many processors). Similarly to the benchmark exercise on the last sheet, use vectors of increasing size. What do you observe?

Exercise 4. (Parallelization of the sparse matrix type)

Now we still need to parallelize the `SparseMatrix` class. As we store our matrix in the CSR format, we parallelize the matrix by distributing the rows to the processes. To this end, expand the interface of your class by adding the following functions:

- Two constructors that take a partition for the rows of the matrix and are otherwise analoguous to the existing sequential constructors:

  ```
  explicit SparseMatrix(ContiguousParallelPartition row_partition,
                        int global_columns,
                        std::function<int(int)> nz_per_row);
  explicit SparseMatrix(ContiguousParallelPartition row_partition,
                        int global_columns,
                        const std::vector<triplet_type>& entries);
  ```

- Two constructors that take an MPI communicator and are otherwise analoguous to the existing sequential constructors:

```
explicit SparseMatrix(MPI_Comm communicator, int local_rows,
                      int global_columns,
                      std::function<int(int)> nz_per_row);
explicit SparseMatrix(MPI_Comm communicator, int local_rows,
                      int global_columns,
                      const std::vector<triplet_type>& entries);
```

- A getter function for the row partition:

```
const ContiguousParallelPartition& row_partition() const;
```

All constructors that are not explicitly parallelized should take `MPI_COMM_SELF` as MPI communicator. Make sure your parallel matrix type passes the tests.

Exercise 5. (Exchange Pattern and Exchange Data)

Before we can parallelize the matrix-vector product, we need two helper classes that facilitate communication and the exchange of data across processes. The matrix-vector product can not be calculated locally on each process, as we might need some vector entries belonging to a different process, according to the sparsity pattern of the matrix. In this case, we call the remote process a neighbor of the local process.
In the following, you are allowed to assume that your sparse matrix is symmetric, i.e. if process q is a neighbor of process p, process p is also a neighbor of process q.

a) Write an `ExchangePattern` class that stores the ranks of the neighboring processes and the global data indices of data that has to be send or received during the computation of the matrix-vector product for the local process. This class has the following interface:

   - A constructor that takes a standard vector of the neighboring processes, global receive and send indices per neighbor process. Assume that the neighboring processes and all send or receive indices per neighbor process are sorted in ascending order.

   ```
   ExchangePattern(std::vector<int> neighboring_processes,
                   std::vector<std::vector<int>> receive_indices,
                   std::vector<std::vector<int>> send_indices);
   ```

   - Getter functions for the neighboring processes, receive and send indices:

   ```
   const std::vector<int>& neighboring_processes() const;
   const std::vector<std::vector<int>>& receive_indices() const;
   const std::vector<std::vector<int>>& send_indices() const;
   ```

   Additionally, write a free function that can create an exchange pattern for a given sparse matrix and the column partition, i.e. the partition of a vector to be multiplied with the matrix. Make sure that the `neighboring_processes`, `receive_indices` and `send_indices` that are passed to the `ExchangePattern` constructor are unique (to not contain double entries) and are sorted in ascending order.

```
template<typename T>
inline ExchangePattern create_exchange_pattern
                       (const SparseMatrix<T>& matrix,
                        const ContiguousParallelPartition& column_partition)
```

b) The second helper class will help you handle the actual data which has been exchanged between the local process and remote processes. It needs to know the exchange pattern associated with your parallel sparse matrix (and parallel vector).

   Write a class `ExchangeData` with the following interface:

   - A constructor that gets an `ExchangePattern` as well as a vector of the data which has been received by remote (neighboring) processes. The constructor simply stores a reference to exchange pattern and the passed data in members of the class:

     ```
     explicit ExchangeData
                  (const ExchangePattern& exchange_pattern,
                   std::vector<std::vector<T>> data_per_neighboring_process);
     ```

   - A get function, which returns the correct data to `global_index` which was send by `owner`:

     ```
     const T& get(int owner_rank, int global_index) const;
     ```

     The implementation of this function can use the reference to the stored exchange pattern to figure out which neighbor has the given `owner_rank` and where to find the entry with the given `global_index`.

   Additionally, write the following free function that creates an `ExchangeData` object from a parallel vector and an `ExchangePattern`:

   ```
   template<typename T>
   ExchangeData<T> exchange_vector_data
                (const ExchangePattern& exchange_pattern,
                 const Vector<T>& vector)
   ```

   This function should exchange the entries of the vector which is to be multiplied with the matrix to the processes owning the corresponding matrix rows.

Exercise 6. (Parallelization of the sparse matrix-vector product)

Now that we have implemented the helper classes `ExchangePattern` and `ExchangeData` we can start to implement the parallel matrix-vector product. The general idea is the following:

- Each matrix stores an instance of `ExchangePattern`. This pattern depends on the sparsity pattern of the matrix (which we assume to be constant during the lifetime of the matrix) and the parallel partition of the vector to be multiplied with. This partition can be interpreted as a partition of the columns of the matrix.
- When a matrix-vector product is to be executed, we first get the exchange pattern that is stored in the matrix. It is assumed that the exchange pattern has been created for the correct parallel partition of the vector.

- The exchange pattern is used to exchange the entries of the vector to be multiplied via `exchange_vector_data`.
- Whenever we need to access an entry of the vector during the multiplication that is not owned by the local process, we will use the function `ExchangeData::get()` to retrieve the value that has been received during the exchange step.

a) Modify the `SparseMatrix` and add the following public member functions:

```
void initialize_exchange_pattern(
  const ContiguousParallelPartition& column_partition);

const ExchangePattern& exchange_pattern() const;
```

Where the first function uses `create_exchange_pattern` to create an exchange pattern for the matrix object and the column partition that is being passed. Since we assume our matrix to be symmetric we can also assume that the passed column partition is the same as the row partition that is already stored within the matrix. The created exchange pattern should be stored in a private data member of the matrix. The second function returns the exchange pattern that is currently stored within the matrix. It is not valid to call this function before `initialize_exchange_pattern` has been called.

b) Modify the `multiply` function that performs the matrix-vector function to perform a parallel product as described above.

A C++ Finite Difference solver in Python

Exercise 7. (Export to Python)

In this exercise we will export the whole functionality of the libraries we have written so far to python, so that applications that are solving specific poisson equations with finite differences can be written easily in python.

a) First we need to install `pybind11` on your computer. We will use the cross-platform package manager `vcpkg` (`https://github.com/microsoft/vcpkg`) to install third-party libraries. To get started using vcpkg follow these instructions:

- **Linux and macOS:** Open a terminal and run

  ```
  cd <WHERE_YOU_WANT_VCPKG_TO_BE_INSTALLED>
  git clone https://github.com/Microsoft/vcpkg.git
  cd vcpkg
  ./bootstrap-vcpkg.sh
  ```

  to build vcpkg.

- **Windows:** Open a PowerShell and run

  ```
  cd <WHERE_YOU_WANT_VCPKG_TO_BE_INSTALLED>
  git clone https://github.com/Microsoft/vcpkg.git
  cd vcpkg
  .\bootstrap-vcpkg.bat
  ```

  to build vcpkg.

Now that vcpkg has been build, we can use it to install `pybind11`:

```
./vcpkg install pybind11
```

b) Add a new file `python/module.cpp` to your repository that contains the `pybind11` entry point `PYBIND11_MODULE` and all the exporting code. Additionally modify the root `CMakeLists.txt` file so that `pybind11` is found and a python module named `pmsc` is build. To make sure CMake can find your installation of pybind11, we have to provide an additional parameter to CMake when configuring your project:

```
cmake .. -GNinja -DCMAKE_BUILD_TYPE=Debug -DCMAKE_TOOLCHAIN_FILE="<
    WHERE_YOU_WANT_VCPKG_TO_BE_INSTALLED>/vcpkg/scripts/
    buildsystems/vcpkg.cmake"
```

c) Add code to export wrapper functions for `MPI_Init` and `MPI_Finalize`. When calling `MPI_Init`, you can pass `nullptr` for both arguments `argc` and `argv` so that you do not need to pass those parameters from Python.

d) Add code to export the linear algebra types `Vector` and `SparseMatrix` and all the operations to python. Use the global type definition `scalar_t` as template parameter for the types. No other template instantiations need to be exported.

e) Add code to export the solver library (solvers, preconditioners and utility types like `StopReason`) to python. To simplify exporting the functionality to python, change the `set_preconditioner` member function of `IterativeSolver` to take a `std::shared_ptr<Preconditioner>` instead of a `std::unique_ptr<Preconditioner>`.

f) Add code to export the grid library with all its types (`Point`, `MultiIndex`, `RegularGrid` and `GridFunction`) to python. Make your python module link to the CMake target `grid_2d` so that your python module will support 2-dimensional types.

g) Add code to export the IO functions for VTK output to python. Since `std::filesystem::path` is not supported by `pybind11` by default, you have to add a small helper function (like in the lecture slides) that takes a `std::string` and than forwards this to the `write_to_vtk` functions.

h) Add code to export the `assemble_poisson_matrix` to python.

i) Write a simple python script that does the following:

- Import your python module
- Call your exported MPI init function
- Set-up a simple 2D grid
- Assemble a discretization of the Poisson equation using boundary conditions and a right-hand-side given as Python functions. You can use the same boundary conditions and right-hand-side as described on the last exercise sheet or try different ones.
- Create a grid function to hold the final result.
- Create and set-up a solver with preconditioner. Solve the assembled equation system
- Write the resulting grid function into a VTK file
- Call your exported MPI finalize function