Dr. Ole Klein, Stefan Meggendorfer
IWR, Heidelberg University

Exercise Sheet 11

Exercise 1: C++ Quiz (5 points)

Here are some additional questions from https://cppquiz.org for you to answer:

Question 1: https://cppquiz.org/quiz/question/219 (variadic function template)

Question 2: https://cppquiz.org/quiz/question/131 (types of initialization)

Question 3: https://cppquiz.org/quiz/question/109 (most vexing parse)

Question 4: https://cppquiz.org/quiz/question/42 (initializer list constructor)

Question 5: https://cppquiz.org/quiz/question/48 (async and futures)

Question 1 revisits variadic function templates. Questions 2 through 4 are about the different kinds of initialization and construction, and certain details that might be surprising. Question 5 is about the futures returned by `std::async`, and how their behavior differs from that of other futures.

Exercise 2: Message Passing Interface (MPI) (10 points)

In this exercise we will train using external libraries and performing parallel computations. Use the *Message Passing Interface* (MPI)[1] to calculate matrix-vector products in parallel.

We will use OpenMPI, but you are free to use any other available MPI software package you like. If you do not have MPI already installed on your machine, you can:

- install it using your package manager (it should be called `openmpi` or similar)
- manually download OpenMPI from https://www.open-mpi.org/software/ompi/v4.1/

In the latter case, extracting and installing is done by the commands:

```
shell$ tar -xvzf openmpi-X.Y.Z.tar.gz
shell$ cd openmpi-X.Y.Z
shell$ mkdir build
shell$ cd build
shell$ ../configure --prefix=/where/to/install
shell$ make all install
```

For more details about the installation process, you can have a look at the provided `INSTALL` file or online at https://www.open-mpi.org/faq/.

If you have never used MPI before, it might be helpful to compile and run a hello world program first, like this C-style example on https://mpitutorial.com/tutorials/mpi-hello-world/.

MPI-based programs have to include a header file called `mpi.h`. They also have to initialize the MPI framework before it can be used, and shut it down before the program finishes. The MPI interface is C code, and may therefore look unfamiliar to you. There are named constants, all uppercase, and functions, with uppercase initial letter:

- Startup/shutdown functions: `MPI_Init`, `MPI_Finalize`
- Communication functions: `MPI_Send`, `MPI_Receive`, `MPI_Alltoall`, `MPI_Barrier`, ...

[1] https://en.wikipedia.org/wiki/Message_Passing_Interface

- Communicators: `MPI_COMM_WORLD`, `MPI_COMM_SELF`
- Data types: `MPI_INT`, `MPI_FLOAT`, `MPI_DOUBLE`, ...
- Reduction operations: `MPI_SUM`, `MPI_PROD`, `MPI_MAX`, ...

Here are some resources you might find helpful, an example program, a condensed overview of commands, and the documentation page:

- https://en.wikipedia.org/wiki/Message_Passing_Interface#Example_program
- https://www.boost.org/doc/libs/1_75_0/doc/html/mpi/c_mapping.html
- https://www.mpi-forum.org/

To compile your code use

```
mpic++ -o application application.cc
```

This is a wrapper around the C++ compiler that takes care of MPI-related stuff. All flags you otherwise would pass to your C++ compiler (like warnings or optimization), can be be passed to `mpic++` as well.

To run your code use

```
mpirun -n 4 ./application
```

where the number 4 stands for the number of processes you would like to start. Take care that you don't accidentally crash your system by invoking too many processes.

In thread-based applications, all the threads share the same memory space, and the main issue is mutual exclusion. In applications based on message passing, however, each process has its own memory space. This means values that are needed for some local computation may need to be sent by one process and received by another.

Write one of the following parallel matrix-vector products, assuming an $n \times n$ matrix with n divisible by the number of processos, and the input and result vectors each distributed in chunks of equal size:

(a) *Horizontally distributed matrix (whole rows).* The i-th entry of the result vector is on the same process as the i-th row of the matrix, so assembling the output is easy, but the required components of the input vector are mostly non-local and have to be communicated.

(b) *Vertically distributed matrix (whole columns).* The j-th entry of the input vector is on the same process as the j-th column of the matrix, so local multiplication is easy, but the components of the output vector have to be accumulated from across all processes.

Which of these two versions is easier to implement? Which, do you think, is more efficient? Choose carefully. Note that there is no need to communicate individual entries when you want to send a whole subvector: the memory layout of `std::vector` is contiguous, and there is even a `data` method that provides access to the underlying raw memory. Make use of that. Receive anything you communicate in a local scope and discard it after using it: in real applications you often don't have the space to keep all intermediate data around (that's one of the reasons to parallelize, after all).

Exercise 3: Message Passing, Part II (10 points (bonus))

Implement the version of parallel matrix-vector multiplication you did not choose in the previous exercise. Determine which of your implementations is faster (for large dimension n), and how the required time scales with the dimension in each version.

Exercise 4: Variadic Templates: Matrix Polynomials (5 points)

In the lecture we discussed the evaluation of polynomials using variadic templates. The recursive function definition allowed using a different type for each of the coefficients, but was restricted to `double` as the type of the variable x and the resulting function value $p(x)$. We would like to make this

function more general step by step, until we are able to evaluate matrix polynomials and polynomial matrices.

A *matrix polynomial*[2] is a polynomial where the scalar variable x is replaced with a square matrix A:

$$p(A) = a_0 I + a_1 A + a_2 A^2 + a_3 A^3 + \dots$$

Any square matrix A commutes with itself, of course, so that integer matrix powers are well-defined. The neutral element of multiplication I (the identity matrix) has to be specified explicitly in this case, while it is normally omitted in ordinary, scalar-valued polynomials.

A *polynomial matrix*[3] is a matrix with polynomials as entries, which can also be written as a (scalar) polynomial with matrices $A_0, A_1, \dots$ as coefficients:

$$p(x) = A_0 + A_1 x + A_2 x^2 + A_3 x^3 + \dots$$

These matrices don't have to be square, but they need to be of compatible dimensions, i.e., each an $n \times m$ matrix, with the same n and m across all the matrices.

Proceed as follows:

(a) Take the function definition of the lecture and introduce a template parameter `X` for the type of the first argument x. The resulting return type is now a function of `X`, `T`, and the remaining types `Ts...`, and is not known a priori: it is certainly not simply `X`, because a polynomial with integer argument and real coefficients has to produce real function values. How can you specify the correct return type?

(b) The function should now be able to evaluate polynomial matrices. Test your implementation with $n \times n$ matrices of small size, say $n = 3$, and make sure that the results are what you expect. Each entry of the resulting matrix is the evaluation of a separate ordinary, scalar-valued polynomial, so this should be straight-forward.

(c) Expand the implementation so that it can also handle matrix polynomials. In this case the function has to compute matrix powers, and it becomes especially important to use Horner's method. Handling matrices as arguments requires the explicit creation of identity matrices, see the definition above. How can you make sure that your function remains usable for ordinary polynomials with real x? Use a matrix class where the dimensions are defined through template parameters, so that you don't have to consider the correct size when creating identity matrices.

(d) Test your implementation on matrix polynomials to make sure that everything works correctly. How does the computation time depend on the size of the matrices, and how on the number of coefficients? Create a variant that uses the less efficient evaluation strategy from the lecture. How does that influence the time that is needed for the computations?

Exercise 5: Custom Range-Based Loops (5 points)

Create your own range-based `for` loops for numerical vectors. Take one of our vector classes, and write code that provides the additional functionality. The vector class itself should not be modified (treat it as if it was from some external library). Instead, use the public interface of the class.

(a) Write an iterator class with the following functionality:

- A constructor taking a vector object. The resulting iterator belongs to this object and iterates over it.
- Pre-increment and pre-decrement operators (`operator++` and `operator--`). These shift the iterator by one position.
- A dereference operator `operator*`, which returns the entry the iterator refers to.
- A not-equal-to operator `operator!=`, for comparison against the end of the vector.

[2] https://en.wikipedia.org/wiki/Matrix_polynomial
[3] https://en.wikipedia.org/wiki/Polynomial_matrix

The iterator has to store its position, the current index is a natural fit. You also need to encode iterators with invalid state, e.g., by using the length of the vector as stored index.

(b) Provide free `begin` and `end` functions that receive a vector object and return an iterator object, pointing to the first entry for the former, and having an invalid state for the latter.

(c) Test your implementation to make sure that it is working, e.g., by printing the vector components in order. Then, provide a `const` version of the class and functions, so that `const` vectors work as well. Use the appropriate standard version of the range-based for loop in each case.