Universität Heidelberg
Institut für Technische Informatik

Robert Strzodka
Application Specific Computing

Parallel Algorithm Design

Term 2021 (Winter)

Exercise 1

- **Submit electronically as one tar(or tar.gz) file inlcuding C++ source files, CMake build system files and an report(PDF) to Moodle until Thursday, 04.11.2021 14:00**
- **Include names on the top of the sheets.**
- **A maximum of three students is allowed to work jointly on the exercises.**

Reading assignment

G. Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities," AFIPS Conference Proceedings, 30:483–485, 1967.

Amdahl's Law on Wikipedia.

J. Gustafson, "Reevaluating Amdahl's Law," Communications of the ACM, 31(5), pp. 532-533, 1988.

Gustafson–Barsis's Law on Wikipedia.

1.1 Allocation and copying (25 points)

Implement the function

```
std::vector<int> matVec(
    const std::vector<int>& A, const std::vector<int>& x);
```

that multiplies matrix `A` (row-major storage) with vector `x` and returns the locally created result `y`. Because the STL vector supports move semantics, no unnecessary copies are made (be sure to compile with at least C++11 support). Do not assume `A` to be a square matrix; from the sizes of the `std::vectors` `A` and `x` you should be able to derive the number of columns and rows in `A`.

The function is convenient to use, but it invariably allocates memory for the result, which may not be optimal. Implement a second variant which permits the caller to preallocate a vector:

```
void matVec(
    std::vector<int>& y,
    const std::vector<int>& A, const std::vector<int>& x);
```

In these benchmarks, what do you measure primarily? Draw a graph based on the benchmark output obtained with '–benchmark_output csv'.

1.2 Lambdas (25 points)

Lambdas are a very convenient way of specifying function objects. Let us say we want to still combine `A` and `x` into a new vector `y`, but instead of multiplication we want to use the xor operation, and instead of summation we want to count how many xor-products per row are equal to a certain given mask, e.g. `0x0F`.

Implement a version of `matVec()` that in addition to `A` and `x` takes two function objects to represent the binary operation (addition or xor operation) and the reduction (addition or masked counting). Then in `main()` specify these function objects using lambdas. Test on a small `A`, for example 8×4, and choose some initial values such that the above mask actually appears for multiple xor-products.

1.3 `std::async()` (40 points)

Now let use further generalize by defining a function `matVecIt()` which does not accept the entire `A`, but rather a begin iterator for the result vector `y` and a begin and an end iterator for the vector `A`, such that we can multiply with a submatrix consisting of some consecutive matrix rows. Having this functionality it becomes easy to implement a parallel version `matVecItPar()`.

`matVecItPar()` takes one more parameter, namely the suggested parallelization factor p. Instead of one call, `matVecItPar()` on a matrix with M rows now performs p asynchronous calls to `matVecIt()` with M/p rows using `std::async()`. Use the matrix from 1.1, check that you get the same result `y`, and compare the execution time for $p = 1, 2, 4, 8, 16$.