High-Performance Computing

Exercises

Prof. Dr. Jan Dünnweber

Research Unit on Distributed Systems and Operating Systems
Faculty of Computer Science and Mathematics
Regensburg University of Applied Sciences

Process Management

1. Get back to your `data.Matrix`-code and create an *abstract* class `Process` for managing parallel processes.
2. Allow subclasses of the `Process`-class to refer to a matrix and to a computation rule (specified via constructor parameters).
3. Provide a `complete()`-method in the `Process`-class, allowing the caller to coordinate processes via a barrier (implemented as a `CountDownLatch`)

Data Partitioning

1. Extend your `Matrix`-class by *efficient* `equals()`, `clone()` and `hashCode()`-implementations, i. e.,
   - `equals()` should terminate once the *first* unequal elements are detected.
   - `clone()` should copy matrices row-by-row using either `Collections.copy()` or `System.arraycopy()`.
   - `hashCode()` should *preferably* yield *different* results for unequal matrices *without* processing every single matrix element. One possible way to achieve this is computing the *trace* of the matrix.
2. Implement `processUL()` and `processLR()`-methods which (analogously to the `process()`-method from the last lab) process only the upper-left or the lower-right triangular submatrix of a square matrix.

note:
you can ignore the fact that `trace()`, `processLR()` and `processUL()` will not work correctly for rectangular matrices (which have no main diagonal). However, ensure that they won't throw `IndexOutOfBoundsExceptions` for any matrix.

(process only the largest enclosed square matrix!)

Testing the Decomposition

1. For testing purposes write an `experiments.Duplication`-class that implements the `Function`-interface such that the `compute()`-method duplicates all matrix elements.
2. Write two new methods inside the `experiments.Parallel`-class:
   - `processMatrix` takes a `Matrix` and a `Function` parameter and maps all elements sequentially in the overridden `body()`-method of a single `Process`-Instance. The return value should be the `Process`' runtime.
   - `processTriangular` gets the same job done by a team of two processes, one mapping the upper-left elements, while the other maps the lower-right part.
3. Implement a **JUnit** test case `decompositionTest( )` that creates a random `test`-matrix, copies it using the `clone()`-method and duplicates all elements of both matrices: `test`-elements via `processMatrix()` and `copy`-elements via `processTriangular`. Prove that the decomposition works by comparing the result matrices via the `equals( )`-method.

Measuring Scalability

1. Move the `compute()`-method that we implemented in the last lab for repeatedly computing the identity function to generate synthetic load into a separate `Identity`-class inside the package `experiments`.
2. Create a 100×100 random matrix and run the sequential `processMatrix()`-method simulating load via `experiments.Identity` and save the computation time T_1.
3. Map the (*unchanged*) result matrix again by running `experiments.Identity` in parallel on its upper-left and its lower-right part using the `processTriangular()`-method and save the computition time T_p.
4. Compute the measured speedup $S(p)$: Divide the sequential runtime by the parallel runtime and output the result.
5. Determine the *efficiency* of your platform as $S(p)/p$.