MISA++: a standardized interface for automated bioimage analysis - Supplementary text

Ruman Gerst\textsuperscript{1,2}, Anna Medyukhina\textsuperscript{1} and Marc Thilo Figge\textsuperscript{1,3,*}

\textsuperscript{1}Applied Systems Biology, Leibniz Institute for Natural Product Research and Infection Biology – Hans-Knöll-Institute, Jena, Germany
\textsuperscript{2}Faculty of Biological Sciences, Friedrich-Schiller-University Jena, Germany
\textsuperscript{2}Institute of Microbiology, Faculty of Biological Sciences, Friedrich-Schiller-University Jena, Germany
\textsuperscript{*}Corresponding author. Email address: thilo.figge@leibniz-hki.de

January 16, 2020

1 C++ framework

<table>
<thead>
<tr>
<th>Library</th>
<th>Version</th>
<th>URL</th>
<th>Author/Citation</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP</td>
<td>4.5</td>
<td>\url{<a href="https://www.openmp.org/%7D">https://www.openmp.org/}</a></td>
<td>[1]</td>
</tr>
<tr>
<td>Boost Libraries</td>
<td>1.67</td>
<td>\url{<a href="https://www.boost.org/%7D">https://www.boost.org/}</a></td>
<td>Beman Dawes, David Abrahams, Rene Rivera; Boost Community</td>
</tr>
<tr>
<td>SQLite</td>
<td>3</td>
<td>\url{<a href="https://www.sqlite.org/%7D">https://www.sqlite.org/}</a></td>
<td>SQLite Consortium</td>
</tr>
<tr>
<td>OME-Files</td>
<td>0.5.0</td>
<td>\url{<a href="https://www.openmicroscopy.org/%7D">https://www.openmicroscopy.org/}</a></td>
<td>[2, 3]</td>
</tr>
<tr>
<td>OpenCV</td>
<td>3.2.0</td>
<td>\url{<a href="https://opencv.org/%7D">https://opencv.org/}</a></td>
<td></td>
</tr>
<tr>
<td>JSON for Modern C++</td>
<td>3.5.0</td>
<td>\url{<a href="https://github.com/nlohmann/json%7D">https://github.com/nlohmann/json}</a></td>
<td>Niels Lohmann</td>
</tr>
<tr>
<td>CMake</td>
<td>3.12</td>
<td>\url{<a href="https://cmake.org/%7D">https://cmake.org/}</a></td>
<td>Kitware Inc.</td>
</tr>
</tbody>
</table>

\textbf{Table S1:} C++ framework dependencies.
Fig. S1: MISA++ organizes workloads in a DAG. It consists of two types of nodes: tasks (blue) and dispatchers (red). Dispatchers are responsible for creating the graph structure by iteratively instantiating direct neighbor nodes (white).
1.1 Project development

Developing a project with MISA++ involves creating a library that has a public application programming interface (API) that provides basic metadata, and a DAG root node. A separate project then wraps the DAG root node into an executable. This separation between executable and dynamically linked library makes it possible for other projects to re-use DAG.

A MISA++ project consists of an project file for the CMake build system (see Listing S1), a package configuration file for CMake (see Listing S2), a global API function that returns metadata (see Listing S3), a C++ class that acts as data interface (see Listing S7), multiple node classes (see Listing S6), and a DAG root C++ class (see Listing S8). An example project structure can be found in Fig. S3.

Additional guides for project creation can be found online (https://applied-systems-biology.github.io/misa-framework/cpp-framework/development/).
Fig. S3: Example project structure. We recommend a clear separation between API headers (folder include), and implementation code (folder src). The CMakeLists.txt project file contains instructions for the CMake build system. Our MISA++ CMake helper functions automatically generate files marked with an asterisk if they do not exist. The folder names (here my-module) correspond to the settings in the CMakeLists.txt file.
**Project file**  The *CMake* configuration (see Listing S1) includes the code files into the build process, and sets up the installation operation that copies the generated library, executable, API headers, and additional metadata into the installation path. The *CMake* build system usually requires developers to manually set up the project configuration for API export. *MISA++* includes script helpers that are automatically available within the `CMakeLists.txt` that includes the *MISA++* core library. Developers only need to provide some metadata and organize the project as shown in Fig. S3. The script then automatically introduces the necessary files and settings to allow proper installation and easy inclusion into other projects. This means that any additional files need to be initialized first, which is done by an integrated development environment (IDE), or manually by running following command in the project folder: `mkdir build; cd build; cmake ...`

```cpp
1 cmake_minimum_required(VERSION 3.11) # or higher if required
2 project(my-module VERSION 1.0.0 DESCRIPTION "")
3 
4 # find_package(misaxx-core REQUIRED)
5 # find_package(misaxx-imaging REQUIRED)
6 # Add additional packages if necessary
7 
8 # Add additional link targets if necessary
9 target_link_libraries(my-module misaxx::misaxx-core misaxx::misaxx-imaging)
10 
11 # MISA++ helper script (automatically included by Core Library)
12 
13 # Automatically generate module_info() function if not existing
14 misaxx_with_default_module_info()
15 
16 # Setup CMake for exporting the API (include files, etc.)
17 misaxx_with_default_api()
18 
19 # Generate an executable out of the DAG root
20 misaxx_with_default_executable()
21```

Listing S1: Example `CMakeLists.txt` project configuration file. It includes the source code into the build process and sets variables for the MISA++ CMake helper script.

**Package configuration**  An example package configuration can be found in Listing S2. An automatically generated file is the *CMake* package configuration that contains information about dependency libraries for the compiled *MISA++* project. During the installation, it is copied to a directory that can be read by *CMake* to allow the library to be found with the easy to use `find_package()` function.

```cpp
1 #package_init
2 
3 # This should have the same find_package commands as
4 # the CMakeLists.txt file
5 find_package(misaxx-core REQUIRED)
6 find_package(misaxx-imaging REQUIRED)
7 # Automatically generated
8 if(NOT TARGET my-module)
```

5
Listing S2: Example CMake package configuration file. This file is required to allow other CMake projects to reuse the functions of the current project. It contains information about which dependency libraries CMake has to import.

Global metadata function Each MISA++ should have a global function that returns information about the current software package (see Listing S3). To cite a third-party MISA++ library, developers just have to add its metadata, which is accessible via the dependency API, to the current object. Non-MISA++ software and papers can also be cited by manually creating a metadata object and setting its entries. An example for such advanced citing can be found in Listing S4.

```
#include($CMAKE_CURRENT_LIST_DIR)/my-module-targets.cmake
endif()
```

Listing S3: Example module information. The code defines an API function that returns metadata such as the name, version, or dependencies.

```
namespace my_module {

misaxx::misaxx_module_info my_module::module_info();
}

// Function declaration in .h file
#include <misaxx/core/module_info.h>
#include <misaxx/imaging/module_info.h>

my_module::module_info() {
    misaxx::misaxx_module_info info;
    info.set_id("my-module");
    info.set_version("1.0.0");
    info.set_name("my-module");
    info.set_description("MISA++ Imaging Support");
    info.add_dependency(misaxx::module_info());
    info.add_dependency(misaxx::imaging::module_info());
    return info;
}
```

// Function implementation - usually in a separate .cpp file
```
misaxx::misaxx_module_info misaxx::imaging::module_info() {
    misaxx::misaxx_module_info info;
    info.set_id("misaxx-imaging");
    info.set_version("1.0.1.0");
    info.set_name("MISA++ Imaging Support");
    info.set_description("Support for OpenCV");
    info.add_author("Raman Gerst");
    info.set_license("BSD-2-Clause");
    info.set_organization("Leibniz Institute for Natural Product Research and Infection Biology - Hans Knosel Institute (HKI), Jena, Germany");
    info.set_url("https://asb-git.hki-jena.de/RGerst/misaxx-imaging/");
}
```

// External dependency: OpenCV
```
misaxx::misaxx_module_info opencv_info;
opencv_info.set_id("opencv");
opencv_info.set_name("OpenCV");
opencv_info.set_url("https://opencv.org/");
opencv_info.set_organization("OpenCV team");
opencv_info.set_citation("Bradski, Gary, and Adrian Kaehler. "OpenCV." Dr. Dobb’s journal of software tools 3 (2000). ");
opencv_info.set_license("BSD-3-Clause");
opencv_info.set_is_external(true);
```

// External dependency: LibTiff
```
misaxx::misaxx_module_info libtiff_info;
libtiff_info.set_id("libtiff");
libtiff_info.set_name("LibTiff");
libtiff_info.set_url("http://www.libtiff.org/");
libtiff_info.set_author("Sam Leffler", "Frank Warmerdam", "Andrey Kiselev", "Mike Welles", "Dwight Kelly");
libtiff_info.set_license("BSD");
libtiff_info.set_is_external(true);
```
Listing S4: Example module information with citations. The metadata objects allow developers to properly cite third-party software. The information can be later extracted via MISA++ CLI commands. In this code example, we left out code already explained in Listing S3.

DAG nodes Each DAG node type corresponds to one C++ class. Task nodes (see Listing S6) have a method `work()` that contains the workload that should be executed by the node. From within a node, developers can access the data interface via the `get_module_as<T>()` function. Alternatively, as they are shared pointers, data caches can also be class attributes and assigned during node instancing.

Parameters are created by adding a class attribute of a MISA++ parameter type. The object is then instantiated within a `create_parameters(builder)` function that connects the parameter attribute to a specific location within the current parameter file. This encapsulates raw access to JSON data to prevent errors, and allows the framework to extract the list of expected parameters via a simulated run.

Dispatcher nodes also have a `create_parameters(builder)` function, but specialized functions designed for DAG node creation instead of a `work()` method. As the DAG root is also a dispatcher node, see Listings S8 to S10 for examples how to assign workloads to dispatchers.

Listing S5: Example task node. Data can be accessed through the module interface instance of the currently assigned sample. Data access is encapsulated through memory-efficient and thread-safe proxy objects. In this example, the input image of the current sample is thresholded with Otsu-thresholding. The resulting mask is written as output image.
Listing S6: Example task node with parameters. Parameters are class attributes that are instantiated within the create_parameters() function. This allows MISA++ to extract information about the parameters in a simulated run. The parameter value then can be queried via a query() method.

Data interface class The data interface class manages the data of one sample (see Listing S7). Its attributes are MISA++ caches that either encapsulate input or output data of a specific type. The setup() function is used to link the caches to a subfolder in the sample input or output directory if no link has been already established by a potential parent data interface. Any cache can generate a self-description of its properties (for example file names or XML metadata) that can be passed from an input to an output cache.
m_output_image.suggest_output_location(filesystem, "output", m_input_image.describe());
}

Listing S7: Example module interface. The code defines the input and output caches and links them to locations within the MISA++ filesystem. In this example, an input image and one output image are declared.

**DAG root** The DAG root, just like any dispatcher node, contains methods to instantiate a part of the DAG (see Listing S8). To allow the framework to automatically document the DAG without the need of data, the process is split into two parts: First, all possible tasks and sub-dispatchers are declared and assigned a name. Those "blueprints" are then instantiated in the second function. There can be as many instances of one blueprint as needed, allowing developers to easily model a parallel mapping (see Listing S9).

Any dispatcher (including the DAG root) have additional high-level capabilities of controlling the order of executed tasks (see Listing S10). They encapsulate two concepts: (i) A group of nodes, and (ii) a pipeline or chain of nodes. Developers can group together any set of nodes and model that another task is only executed after all grouped nodes have finished their workload. Chains allow easy creation of pipelines \((N_1, N_2, \ldots, N_{k-1}, N_k)\) where a node \(N_i\) is only executed after \(N_{i-1}\) has finished its workload.

// Class declaration in .h file
#include <misaxx/core/misa_module.h>
#include <my-module/module_interface.h>
namespace my_module {
struct module : public misaxx::misa_module<module_interface> {
  using misaxx::misa_module<module_interface>::misa_module;
  void create_blueprints(blueprint_list &blueprints, parameter_list &parameters) override;
  void build(const blueprint_builder &builder) override;
};
}

// Class definition in .cpp file
#include <my-module/my_task.h>
void my_module::module::create_blueprints(blueprint_list &blueprints, parameter_list &parameters) {
  // Register task "my-task" of type my_task
  blueprints.add(create_blueprint<my_task>("my-task"));
}

void my_module::module::build(const blueprint_builder &builder) {
  // Instantiate "my-task"
  my_task &task = builder.build<my_task>("my-task");
}

Listing S8: Example module DAG root. Tasks, dispatchers and sub-modules are first registered before their actual instantiation. In this example, a task my_task with the name my-task is declared without actually instancing it. Instantiation happens in the build() function where the task is instantiated based on its name. In more complex programs, multiple instances of the same task type can be created for parallelization.

// We assume that m_input_image contains a dynamic amount of slices
// We assume that m_output_segmented contains the same amount of slices as m_input_image
void my_module::module::create_blueprints(blueprint_list &blueprints, parameter_list &parameters) {
  // Register task "segment-2d-layer" of type segment_2d_layer
  blueprints.add(create_blueprint<segment_2d_layer>("segment-2d-layer"));
}

void my_module::module::build(const blueprint_builder &builder) {
  for(size_t i = 0; i < m_input_image.num_slices; ++i) {
    segment_2d_layer &task = builder.create<segment_2d_layer>("segment-2d-layer");
  }

// We assume that m_input_image contains a dynamic amount of slices
// We assume that m_output_segmented contains the same amount of slices as m_input_image
void my_module::module::create_blueprints(blueprint_list &blueprints, parameter_list &parameters) {
  // Register task "segment-2d-layer" of type segment_2d_layer
  blueprints.add(create_blueprint<segment_2d_layer>("segment-2d-layer"));
}

void my_module::module::build(const blueprint_builder &builder) {
  for(size_t i = 0; i < m_input_image.num_slices; ++i) {
    segment_2d_layer &task = builder.create<segment_2d_layer>("segment-2d-layer");
  }

// We assume that m_input_image contains a dynamic amount of slices
// We assume that m_output_segmented contains the same amount of slices as m_input_image
void my_module::module::create_blueprints(blueprint_list &blueprints, parameter_list &parameters) {
  // Register task "segment-2d-layer" of type segment_2d_layer
  blueprints.add(create_blueprint<segment_2d_layer>("segment-2d-layer"));
}

void my_module::module::build(const blueprint_builder &builder) {
  for(size_t i = 0; i < m_input_image.num_slices; ++i) {
    segment_2d_layer &task = builder.create<segment_2d_layer>("segment-2d-layer");
Listing S9: Parallelization mapping example. Developers can instantiate any amount of nodes of the same type. This blueprint-like feature allows dynamic parallelization. In this example, we left out code that has already been explained in Listing S8.

Listing S10: Dependency management example. Any dispatcher comes with powerful tools to further control DAG dependencies without the need of creating a separate dispatcher class. In this example, we left out code that has already been explained in Listing S8.
<table>
<thead>
<tr>
<th>Library</th>
<th>Version</th>
<th>URL</th>
<th>Author/Citation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageJ, ImgLib2</td>
<td>2.0.0</td>
<td><a href="http://imagej.net/">http://imagej.net/</a></td>
<td></td>
</tr>
<tr>
<td>Gson</td>
<td>2.8.5</td>
<td><a href="https://github.com/google/gson">https://github.com/google/gson</a></td>
<td>Google LLC</td>
</tr>
<tr>
<td>Guava</td>
<td>26.0-jre</td>
<td><a href="https://github.com/google/guava">https://github.com/google/guava</a></td>
<td>Google LLC</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>1.5.0</td>
<td><a href="http://www.jfree.org/jfreechart/">http://www.jfree.org/jfreechart/</a></td>
<td>Dave Gilbert</td>
</tr>
<tr>
<td>JFreeSVG</td>
<td>3.3</td>
<td><a href="http://www.jfree.org/jfreesvg/">http://www.jfree.org/jfreesvg/</a></td>
<td>Dave Gilbert</td>
</tr>
<tr>
<td>flexmark-java</td>
<td>0.40.18</td>
<td><a href="https://github.com/vsch/flexmark-java">https://github.com/vsch/flexmark-java</a></td>
<td>Vladimir Schneider</td>
</tr>
<tr>
<td>Bioformats Plugin</td>
<td>6.0.0</td>
<td><a href="http://www.openmicroscopy.org/bio-formats">http://www.openmicroscopy.org/bio-formats</a></td>
<td></td>
</tr>
<tr>
<td>SQLite JDBC Driver</td>
<td>3.25.2</td>
<td><a href="https://bitbucket.org/xerial/sqlite-jdbc">https://bitbucket.org/xerial/sqlite-jdbc</a></td>
<td>Taro L. Saito</td>
</tr>
<tr>
<td>Apache POI</td>
<td>4.0.1</td>
<td><a href="https://poi.apache.org/">https://poi.apache.org/</a></td>
<td>Apache Software Foundation</td>
</tr>
<tr>
<td>ICEpdf</td>
<td>6.2.2</td>
<td><a href="http://www.icesoft.org/">http://www.icesoft.org/</a></td>
<td>Icesoft Technologies Inc.</td>
</tr>
<tr>
<td>Apache Commons Exec</td>
<td>1.4.0</td>
<td><a href="https://commons.apache.org/proper/commons-exec/">https://commons.apache.org/proper/commons-exec/</a></td>
<td>Apache Software Foundation</td>
</tr>
<tr>
<td>Breeze Icons</td>
<td>-</td>
<td><a href="https://github.com/KDE/breeze-icons">https://github.com/KDE/breeze-icons</a></td>
<td>KDE e.V.</td>
</tr>
</tbody>
</table>

Table S2: ImageJ plugin dependencies.
2 ImageJ plugin

2.1 ImageJ plugin

To showcase the capabilities of the MISA++ interface, we developed a plugin for ImageJ that provides a graphical user interface for any MISA++ application. It utilizes the human- and machine-readable documentation that can be generated by any MISA++ application to setup the user interface.

Step-by-step tutorials on how to use the plugin are available online in writing and video form at https://applied-systems-biology.github.io/misa-framework/imagej/step-by-step/

Starting the ImageJ plugin opens a list of available MISA++ applications along with a description. MISA++ applications are either detected automatically or can be added by selecting the executable. From within the application list, users have access to features that allow analyzing data with a single MISA++ application, connect multiple applications into one pipeline, and analyze the output of an application.

2.1.1 Data analysis

To analyze data, users must first setup the list of image samples. Samples are identified by their name and can be either imported from an existing folder that follows the VFS structure or provided via manual input. Images can be imported directly from ImageJ and are automatically converted into a format that is compatible with the cache. For non-image data types and custom caches, users can select the files from the file system. The MISA++ ImageJ plugin automatically creates the required input folder structure and warns users if data is missing. After setting up the data, users can change sample-, algorithm- and application-wide parameters via a graphical user interface. Documentation provided during the parameter declaration is accessible from within the interface. The settings are automatically validated by the plugin. Users can choose to either run the MISA++ application on the current computer or export a data and parameter package for analysis on another computer or server. If the analysis is done on the current computer, the tool will offer to display and evaluate the results.

2.1.2 Pipeline creation

In addition to running a single MISA++ application, our plugin also includes a tool to create pipelines. They are represented in a flowchart where processing steps are MISA++ applications and arrows are dependencies. To create a pipeline, users can add any known MISA++ application to the flowchart. Dependencies can be created by clicking a button that shows possible connections to the selected processing step. With an established dependency, output data can be passed from one step to the input of another step via the data import interface (see section S2.1.1). Another similarity to a single-application analysis is that pipelines can be either run directly from within ImageJ or exported for analysis on another computer.

2.1.3 Result analysis

We included a tool to simplify the further analysis of MISA++ application results. It allows (i) importing result images back into ImageJ, (ii) browsing, summarizing, and plotting of quantification results, and (iii) analyzing the runtime. The graphical user interface displays all available output and input data. Compatible data can be imported back into ImageJ. The exact set of available actions depend on the cache data type and can be extended with custom plugins.

MISA++ applications store quantification results in a standardized format that can carry large amounts of additional metadata like units or localization information about a segmented
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Tested value</th>
<th>MISA++ Java</th>
<th>MISA++ Python</th>
<th>Java Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glomeruli segmentation</td>
<td>Glomerular number</td>
<td>&gt; 0.999</td>
<td>0.976</td>
<td>&gt;0.999</td>
</tr>
<tr>
<td>Glomeruli segmentation</td>
<td>Tissue volume</td>
<td>&gt; 0.999</td>
<td>0.978</td>
<td>0.983</td>
</tr>
<tr>
<td>Cell segmentation</td>
<td>Cell count</td>
<td>&gt;0.999</td>
<td>&gt;0.999</td>
<td>&gt;0.999</td>
</tr>
<tr>
<td>Deconvolution</td>
<td>Distance</td>
<td>0.823</td>
<td>&gt;0.999</td>
<td>0.912</td>
</tr>
</tbody>
</table>

**Table S3**: p-values calculated from pairwise comparing the characteristic algorithm readout distributions via Bootstrap Kolmogorov-Smirnov test ($n_{boots} = 1000$). All p-values are well above the critical value of 5%.

To allow efficient storage of the metadata, an external MISA++ application transforms JSON into an SQLite (https://www.sqlite.org/) database. The user interface includes functions to easily query this database. Users can browse, filter, and export entries in table format. Tables can either be exported into commonly used spreadsheet formats, or further analyzed. We included features to extract of basic descriptive statistics, and plot the results directly from within **ImageJ**.

Any MISA++ application is able to log its runtime with the option detailed per-task logs. Our **ImageJ** plugin comes with a tool to plot the runtime and extract descriptive statistics like the estimated multi-threaded speedup. Developers can use this information to spot performance bottlenecks and further increase the application performance.

### 3 Image analysis

#### 3.1 Glomeruli analysis

```json
{
  "filesystem": {
    "input-directory": "/data/input",
    "output-directory": "/data/output",
    "source": "directories"
  },
  "runtime": {
    "misaxx-ome": {
      "disable-write-buffer-to-ome-tiff": true
    },
    "num-threads": 30,
    "full-runtime-log": true
  },
  "samples": {
    "Kontrolle_Bonn 1a zoom063 zoom 647.12-39-10": {},
    "Kontrolle_Bonn 1b zoom063 zoom 647.13-22-21": {},
    "Kontrolle_Bonn 2a zoom063 zoom 647.11-53-22": {},
    "Kontrolle_Bonn 2b 10.23-40": {},
    "Kontrolle_Ly6G M11a zoom08 zoom a1647.14-26-05": {},
    "Kontrolle_Ly6G M11b zoom08 zoom a1647.13-43-00": {},
    "Kontrolle_Ly6G M1a zoom08 zoom a1647.12-38-06": {},
    "Kontrolle_Ly6G M1b zoom08 zoom a1647.13-05-34": {},
    "Kontrolle_Ly6G M9a zoom08 zoom a1647.11-36-12": {},
    "Kontrolle_Ly6G M9b zoom08 zoom a1647.12-10-14": {},
    "d15 NTN 2a zoom08.15-33-00": {},
    "d15 NTN1a zoom08.15-19-34": {},
    "d15 NTN1b zoom08.13-43-09": {},
    "d15 NTN2b zoom08.14-23-01": {},
    "d15 NTN3a zoom08.16-04-48": {},
    "d15 NTN3b zoom08.16-29-01": {},
    "d7 NTN 1a zoom063 zoom 647.09-40-46": {}.
  }
}
```
Listing S11: Parameter file for MISA++ glomeruli segmentation. The file sets up the VFS, number of threads, samples, and disables OME TIFF writing for consistency with the Python implementation.

Our glomeruli analysis algorithm [6, 7] applies the following steps: The initial preprocessing consists of a median filter that removes noise. Afterwards, the tissue is segmented via a percentile thresholding with subsequent morphological hole-closing operations. A filtering step is then applied to remove false-positive tissue regions via a mean intensity threshold. Glomeruli segmentation requires the minimum and maximum object radius and begins with segmenting the glomeruli in 2D. A morphological top-hat operation is applied to the preprocessed image to segment objects larger than the minimum radius. The glomeruli are then segmented via Otsu thresholding. Objects larger than the maximum size are removed via a morphological opening, while false-positive objects outside the tissue region are erased.

3.2 Cell segmentation

Segmentation and quantification of conidia – fungal spores – can be used to assess the behavior of immune cells [8]. One difficulty in segmenting conidia in 2D microscopy images is that cells might be merged during thresholding (see Fig. S6). A common method to separate merged cells consists of initial smoothing and Otsu thresholding. To separate merged cells, distance transformation is applied. The resulting image is used to find seed points via local maxima and as input for a watershed-algorithm.

The DAG is similar to the tissue segmentation where all 2D planes are segmented first and the resulting masks are accumulated afterwards.

Our MISA++ implementation was over 137 times faster than the Java application and about 4.6 times faster than the Python implementation.

Our test for difference in distribution showed the results to be significantly equal in all tested cases (see Table S3).

3.3 Deconvolution

Deconvolution is a common image restoration technique (see Fig. S7). It is based on the idea that a microscopy image is convolved by a point spread function (PSF). If the PSF is known, an inverse filter operation in Fourier space then can be used to restore the original image. In practice, naive inverse filtering yields usually bad results. There are different tools like DeconvolutionLab2 [9] that implement alternative methods. For our example, we implemented the Regularized inverse filtering algorithm in MISA++ and Python, and used DeconvolutionLab2 for the Java implementation.

Given the fourier-transformation of the input image $Y$, the fourier-transformed deconvolved image $X$ can be calculated via the Regularized inverse filter [9] method:

$$X = Y \cdot \frac{H}{H^2 + L \cdot \lambda \cdot L}$$

(1), where $\lambda$ is the regularization factor, and $L$ is the fourier-transformation of the $3 \times 3$ laplacian kernel. For our analyses, we set the regularization factor $\lambda = 0.001$. 


Fig. S4: Glomeruli segmentation result comparison. For the number of glomeruli, the maximum pairwise difference is 2344 (13.63%), and on average 168.58 ± 412.86 (0.98% ± 2.4%). The tissue volume is at most 7.79 \cdot 10^9 \mu m^3 (3.85%) and on average 5.67 \cdot 10^8 \mu m^3 ± 1.32 \cdot 10^9 \mu m^3 (0.28% ± 0.65%) different. The mean glomerulus diameter is on average 0.42 \mu m ± 0.27 \mu m (0.6% ± 0.39%) and at most 1.39 \mu m (2.01%) different.
**Fig. S5:** Glomeruli segmentation result decile comparison. The zero-line is within the confidence intervals for glomerular number and tissue volume ($n_{\text{boots}} = 100$). The decile differences in diameters are outside the confidence intervals in all tested pairs, but well below image voxel size of $4.063 \, \mu m \times 4.063 \, \mu m \times 5 \, \mu m$.

**Fig. S6:** Conidia (fungal spores) are fluorescently labeled (a). During thresholding, conidia might be merged into one object, so a watershed algorithm is used to separate them. The image contrast was enhanced for visualization. (b) DAG structure of the *MISA++* conidia segmentation tool. A sample consists of multiple 2D experiments that are segmented in parallel. Afterwards, the cells are quantified.
The DAG consists of a convolution task with a known PSF, followed by a simulated deconvolution with the same PSF.

Our MISA++ implementation was about 17 times faster than the Java application and about 5.8 times faster than the Python implementation.

To allow comparison between different implementations, we calculated the Wasserstein distance between each deconvolved and its original image. Before calculating the distance, we first standardized the images to have a average value of zero, and a standard deviation of one. Each 2D image was converted to an 1D vector. Our test for difference in distance distribution showed the results to be significantly equal in all tested cases (see Table S3).

### 3.4 Single-operation benchmarks

The single-operation benchmark applications runs following tests:

- **canny** A canny edge detection algorithm $Canny(\sigma_{\text{gaussian}}, T_1, T_2)$ with gaussian pre-processing $\sigma_{\text{gaussian}} = 1$, and thresholds $T_1 = 0.1$ and $T_2 = 0.2$

- **fft-iff** The input image is transformed into its fourier-space representation and transformed back into real space

- **io** Loading the input image and saving 7 output images as LZW-compressed 32 bit float TIFF files

- **median** Median-filtering with a $21 \times 21$ square mask

- **morphology** Dilation with a circle mask of radius 15

- **otsu** Otsu-thresholding

- **percentile** $65th$ percentile thresholding

- **wiener2** Applying a Wiener deconvolution according to the Matlab implementation (see https://de.mathworks.com/help/images/ref/wiener2.html)
Fig. S8: Single-operation benchmark results. Results marked with a (*) were executed without additional parallelization offered by OpenCV and Python image libraries. C++ was the fastest in 5 out of 8 cases.

Table S4: p-values calculated from pairwise comparing the distributions of the Wasserstein distances between original and convolved microbenchmark images. We use a Bootstrap Kolmogorov-Smirnov test ($n_{\text{boots}} = 1000$).
Fig. S9: Single-operation benchmark shift plots. All pairwise differences are within confidence intervals only in 5 out of 7 tested operations.
In our single-operation benchmark, MISA++ was the fastest in 5 out of 8 cases. The cases where C++ was not the fastest implementation were Fourier transformation, TIFF input and output, and percentile thresholding. We cannot explain the differences in TIFF IO performance, as both MISA++ and Python use libtiff (http://libtiff.org/) for TIFF reading and writing with LZW compression enabled.

The Fourier transform performance can be explained by Java creating a differently sized frequency space representation: ImgLib2 offers Fast Fourier Transform (FFT) and Inverse Fast Fourier Transform (iFFT) algorithms, but does not allow developers to specify the size of the resulting frequency image. In our tests, ImgLib2 produced FFT outputs with smaller dimensions than the input image, while OpenCV and NumPy were instructed to generate outputs of the same size as the original image.

Lastly, the low C++ in percentile thresholding performance can be explained by a naive custom implementation that first extracts all pixels and then applies a sorting algorithm. The performance can be optimized by applying partial sorting algorithms, as percentile extraction only requires a subset of conditions that are created by sorting.

As an addition to a versus-language comparison, we also tested the influence of additional optimization that is usually enabled by default for any image analysis library. Our results show differences between optimized and unoptimized workloads. The speed impact depends on the operation and the image library.

To allow testing if the results are equal across implementations, we calculated the Wasserstein distance between output and input image for each sample. We standardized the input image, so it has a mean value of zero and standard variance of one. We standardized the output images for non-thresholding algorithms (Dilation, FFT/iFFT, median filtering, Wiener deconvolution). For thresholding, we instead normalized the image by its maximum value. The reason behind this is that standardization transforms from binary value space to real value space, which distorts thresholded results.

We then compared the distributions via pairwise Bootstrap Kolgomorov-Smirnov test ($n_{\text{boots}} = 1000$). Our results show that not all distributions were significantly equal ($p \gg 0.05$; see Table S4). Visualizations of the differences via decile shift plots are consistent with the test results (see Fig. S9). We address the observed differences below.

**FFT/iFFT** Evaluation of the result images revealed that the C++ results were shifted, explaining the large difference between C++ and other implementations. While the results are different, the program still applied a proper inverse FFT (iFFT) on previously generated frequency-space data. For real-world implementations (see Deconvolution example), the code is modified to extract the correct real image from periodic frequency space. We still see the performance comparison as valid, as equivalent operations were applied — just shifted by period.

While the difference between Java and Python results are significant ($p < 2.22e-16$), they decile differences are at most $\pm 2.7 \cdot 10^{-3}$. Considering implementation details like output sizing, and value scaling,

**Median filter** Our results indicate that the Java median filter implementation produces a different output. Further investigation revealed that imglib2 converts the image to an 8-bit representation, while C++ and Python implementations do not convert the floating point images. Although there are significant differences, the algorithms still have to apply similar operations for each neighborhood.

**Wiener deconvolution** The Wiener deconvolution results were significantly different between C++ and the other implementations. We implemented this operation manually and re-confirmed that the operations are equivalent. To prevent zero-division errors, we added
a constant to local variance component. This constant is very small (around $10^{-6}$) and therefore can cause floating point errors depending on how the code is compiled into machine code.

### 3.5 Impact of additional optimizations

Python and C++ image libraries allow developers to disable per-operation optimizations such as parallelization, or GPU computing. To test the influence of those optimizations, we repeated certain analyses without enabled optimizations. In case of Java image libraries, optimizations are usually enabled by default without an easy way for disabling them, or provided in code as explicit number of threads. Therefore, we did not compare the impact of optimization in Java applications.

Testing the impact of optimization on our three composite algorithms (see Fig. S10) shows that optimization can even have a negative impact if the workload is relatively small. For larger workloads, such as the 3D glomeruli segmentation, optimization can reduce the runtime by up to 38 minutes for the Python implementation. Optimization therefore is only reasonable if the processing steps are slow. Additional parallelization overhead, such as thread creation or uploading data to the graphics card, can negatively impact the performance for tasks that already run very fast.

Single-operation benchmarks (see Fig. S8) reveal that additional optimization impacts each operation in a different way.
Fig. S10: Impact of additional optimization on composite algorithms. Results marked with an asterisk are run with disabled optimizations. The impact depends on the workload.
References


