Chapter 6
Interoperability Between Software and Hardware

6.1 Hardware Options for Accelerating Computations

Introduction to Hardware and Software Interoperability
Big image data can require significant processing time. Software and hardware can be exploited to shorten this time. In addition to increasing microscope acquisition speed and exponential data growth, information technology is also rapidly advancing. To leverage this new information technology, one must deal with a diversity of hardware and software interfaces. Software must be written with hardware specifications in mind and must integrate with other existing software. These software engineering activities are driven by our goals not only to shorten computation time but also to minimize the effort of building big data analytic solutions. While WIPP can be deployed on a variety of hardware, its execution performance will depend on the selection of that underlying hardware. We present next the hardware options with their pros and cons.

Advanced hardware options
In general, advanced hardware can accelerate big data processing by minimizing the time for read/write operations, network data movement, and data processing. Based on hardware specifications, minimization of the overall execution time can be achieved by:

(a) Keeping all data in RAM during computations instead of transferring them from hard drives
(b) Utilizing processors with higher clock speeds to increase the number of computations per time unit
(c) Using high bandwidth communication buses to decrease data transfer time
(d) Parallelizing read/write, data movement, and computation operations
(e) Utilizing special computational hardware that is more efficient than available central processing units (CPUs)
(f) Using nonvolatile memory (NVM) express devices with solid-state drives (SSDs) for faster data access

P. Bajcsy et al., *Web Microanalysis of Big Image Data*,
https://doi.org/10.1007/978-3-319-63360-2_6

To implement big data performance acceleration, there are currently four hardware options for principal investigators or small research teams:

1. Access a supercomputer with a very large RAM, fast connectivity, and powerful computational nodes (CPUs and GPUs).
2. Buy a high-end desktop computer with advanced hardware (large RAM, powerful processors, fast buses), and fully utilize this hardware for faster computation.
3. Access multiple computers (either in a cluster or cloud) and exploit them in parallel.
4. Utilize additional computer hardware (e.g., GPUs, FPGAs, or NVM SSDs) designed specifically for efficient computation and fast storage, and then implement custom software for this hardware.

All of the above options require integrating software and hardware by redesigning existing software or designing new software to fully utilize the hardware. We will briefly describe each of the four options.

Access a supercomputer
The supercomputers are available to a small percentage of researchers because of their limited availability. The key difference between the big data and supercomputing communities is the ratio of data size to the number of computations per data point. The primary focus of the supercomputing community is on computations whose results require little input data but a very large amount of computation. Numerical simulations are an example. The big data community, on the other hand, is interested in information extraction and data-driven modeling from very large datasets. Information extraction might not require a large number of computations per data point, but the number of data points is very large. Unlike information extraction, data-driven modeling can also demand a large number of computations per data point. The ratio of data size to the number of computations for big data computations leads us to emphasize data distribution in order to parallelize computations.

Buy a high-end desktop computer
One can invest in a personal computer with advanced hardware (faster processor, more RAM, faster disk access, etc.). This approach is expensive and is limited by the best available hardware. The currently available multicore processors require developing algorithmic implementations that can leverage them.

We illustrate this approach in Fig. 6.1 by drawing a parallel between the throughput of big data processing and the throughput of cars on automobile highways. The colored cars are types of computations, and a highway lane is one processor with its bus connecting the processor to all needed data. Buying a multicore computer to run single-threaded code can be compared to paying for a multiple lane highway and then using only one lane. Our throughput metric is the number of computations per time (or number of cars per time interval). It is apparent that the software affects performance by defining how the big data input is handled by hardware during computation.

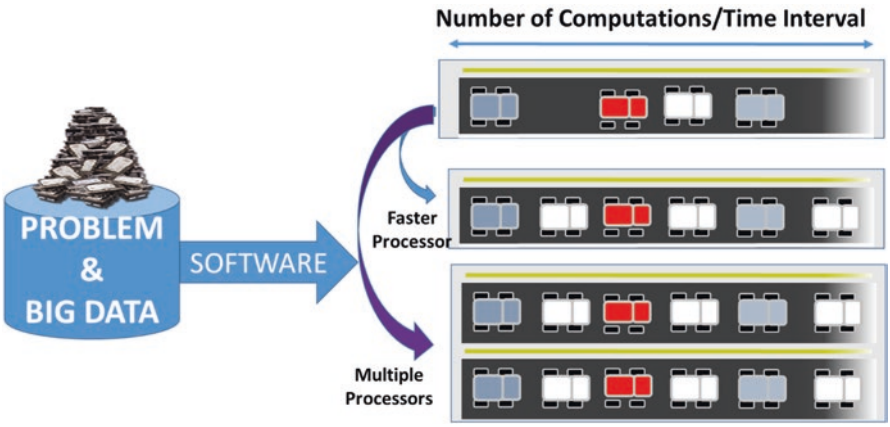

Fig. 6.1 Accelerating big data problem computations by purchasing a faster processor or by redesigning software and buying a computer with multiple processors

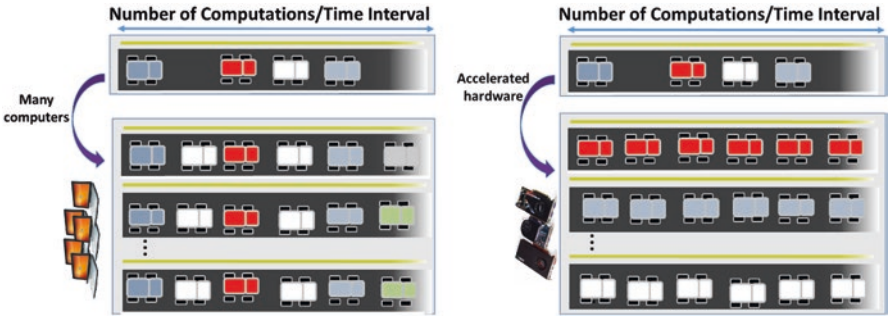

Fig. 6.2 Accelerating big data problem computations by redesigning software to run on multiple computers in parallel (left) or on accelerated hardware and attaching the hardware to a computer

Access multiple computers

Another option for accelerating big data computations is to use multiple computers in parallel as illustrated in Fig. 6.2 (left). This option requires designing algorithmic implementations together with the software that would move parts of the data and parts of the computations to multiple computers and then collect the results on a single computer. Critical components in this case are (a) the decomposition of a computation into steps that can be executed in parallel and (b) the partition of input data into chunks that are needed by each step. We will focus primarily on the Hadoop framework [1] for such algorithmic implementations.

Utilize additional computer hardware

The last option is to use additional specialize hardware for accelerating computations as illustrated in Fig. 6.2 (right). Special computer hardware can perform some computational steps more efficiently than a general-purpose CPU. Figure 6.2 (right)

shows color-coded grouping of implementations per processor (cars per highway lane) since each hardware acceleration including software and hardware is specifically designed to minimize execution time per computational step. Examples of such computer hardware are graphics processing units (GPUs) or field-programmable gate arrays (FPGAs). This option requires writing the algorithmic implementation in a language that is hardware-specific or a language that can be compiled into hardware-specific code. Like the multiple computer option, a critical component is the parallel decomposition of a computation with the accelerated hardware specifications in mind (i.e., bus speed from a computer to a card, available RAM on a card, etc.). The use of nonvolatile memory solid-state devices (NVM SSDs) in the form of standard-sized Peripheral Component Interconnect (PCI) cards might bring as much overall time savings as the GPUs and FPGAs because of the significant time spent reading and writing big image data.

6.2 Implications of Big Data Attributes

In any big data experiment, it is beneficial to estimate the rate of image acquisition and the size of the image collections to be processed within a given time interval. These estimates allow a user to plan hardware purchases and select big data solutions according to attributes of big data.

Attributes of big data
The distribution of big data across multicore processors, multiple computers, and multiple cards with accelerated hardware should be performed with respect to big data attributes. The attributes defining big data are described as four Vs and include:

- Volume
- Velocity
- Variety
- Veracity

Volume is the data size on disk measured in bytes. Velocity is the speed in which data can be accessed, for example, the data acquired by a microscope scanning a slide. Variety refers to structured or unstructured organizations of different types of data (e.g., structured XML key-value pairs or unstructured text from blogs). Veracity reflects whether the data are accurate or not (calibrated vs. uncalibrated microscope images, inaccurate metadata about images).

Some practitioners[1] add additional attributes such as:

- Variability
- Visualization
- Value

[1] https://www.impactradius.com/blog/7-vs-big-data/

These extra attributes are less frequently cited. Variability refers to inconsistent meaning or labeling of data, for instance, two microscopes reporting dimensions of a pixel as "pixel size" or "pixel dimension" (important for data integration). Visualization refers to conveying the meaning in a pictorial or graphical format rather than in a spreadsheet numerical format. Value is the ultimate measure of the information gain after addressing volume, velocity, variety, variability, veracity, and visualization.

Big data solutions

The application-specific big data attributes have implications on storage, network, and computational hardware specifications which lead to solutions that operate from the scale of an imaging lab to the scale of a large IT company. For example, by considering the volume and velocity attributes, a user must plan for big data storage that (a) can handle the current size and the data growth (scale) and (b) can provide a fast access to the data (the input/output operations per second (IOPS)). We will describe the implications of big data attributes on hardware and software at commercial and imaging laboratory scales and the role of interoperability and standards for big data solutions.

Big data solutions at commercial scale

In a commercial space, large IT companies create *hyperscale computing environments*, where the term "hyper" refers to excessive and "scale" refers to data growth.[2] The hyperscale computing environment is achieved by assembling commodity servers for petabytes of data with direct-attached storage (DAS) and storage redundancy at the level of the entire computer/storage unit. In order to serve millions of users with thousands of applications, these environments are reducing storage latency by having Peripheral Component Interconnect Express (PCI-E) flash storage and running analytic engines like Hadoop, NoSQL, and Cassandra.

Big data solutions at imaging laboratory scale

For researchers running big microscopy image experiments, the volume attribute of big data is typically at the scale of terabytes, and the velocity is between 1 MB/s and 100 MB/s (see Chap. 1, Fig. 1.6). This implies that network-attached storage (NAS) and clustered NAS with multiple terabyte-sized storage capacity might be sufficient and cost-efficient. As software for clustering NAS improves to deliver petabyte/parallel file system capability, this solution may meet the requirements for *scaling up* (adding more disks to the same disk controllers). With the growing number of cloud providers, there is also an option of *scaling out* by creating a distributed storage (requesting more nodes in the computer cloud to provide a larger aggregated storage). Finally, the need for collocating data with computational resources suggests an option of bringing data storage and computers into one geographical locations with fast network connectivity, cooling, and sufficient power. This option is provided by commercial vendors and is referred to as *colocation centers*.

[2] http://searchstorage.techtarget.com/podcast/Understanding-stripped-down-hyperscale-storage-for-big-data-use-cases

Interoperability and Standards
The attributes of big data have been one of the main topics discussed in the NIST Big Data Public Working Group (NBD-PWG).[3] This working group has focused on Big Data Definition, Big Data Taxonomies, Big Data Use Cases and Requirements, Big Data Security and Privacy, Big Data Reference Architecture, Big Data Standards Roadmap, Big Data Reference Architecture Interface, and Big Data Adoption and Modernization. The work of NBD-PWG has led to reports that address the interoperability of big data solutions ever since "the world was awash with over 800 exabytes of data and growing" as estimated in 2010 by Thomson Reuters.[4]

6.3 Execution Times of Computation Over Big Image Data

Response time-based classification of computations
Based on the storage considerations for the image data volume and velocity attributes, one must decide on the hardware and software. Processing is classified as either *off-line* or *interactive*. In other words, interactive computations are expected to take the time of a mouse click (i.e., seconds), while off-line computations can take minutes, hours, and days. This classification is focused on a response time to user inputs. For systems designed for interactive, discovery-type, analyses, and image explorations, it is important to:

1. Classify computations based on their execution time requirements.
2. Decide on strategies to meet the time requirements.
3. Support the chosen strategy by evaluations of computational complexity.
4. Collect measurements of actual execution times to validate the system performance.

In the next two subsections, we focus on (1) meeting execution time requirements and (2) estimating and measuring execution time over big image data.

6.3.1 Meeting Execution Time Requirements

Local vs. distributed computing
To meet performance requirements, analyses can be executed on a local computer or on a set of distributed computational nodes. For local analyses, data are transferred to a local computer, and computations are constrained to a local desktop. For distributed analyses, data chunks are sprayed across multiple computer cluster or cloud nodes. Computations are executed by processing data chunks followed by merging partial results into an overall result. If advanced hardware is available on local or distributed computational resources, then the data must be transferred to it

[3] https://bigdatawg.nist.gov/
[4] http://archive.annual-report.thomsonreuters.com/2010/

(e.g., to GPU or FPGA memory). The key aspect of these processing options is to co-localize big data with the most processing power. The execution time depends on the time to move data, the availability of processing resources, and the software design for fully utilizing each hardware device.

On-demand computing
Another aspect of the performance requirements is the number of concurrent users. Users launch computations in an unpredictable pattern on web systems like WIPP. If the computational resources are busy, then computations will wait in a queue and cause long execution times. During these "on-demand" computations, the hardware and software solutions must have access to additional computational resources to meet performance requirements. The resources could be plug-and-play hardware devices or new virtual machines (i.e., elasticity of cloud computing).

Client-server computing as a type of on-demand computing
In the context of client-server web systems, the web server accesses distributed computing resources to execute analyses (see Fig. 6.3). The resource allocation for distributed analyses depends on computational demands. It can be elastic as described above. In comparison to execution using distributed computing resources, local analyses can be executed on a client computer or on a main web server. The challenge lies in co-localizing data and processing power. For example, if images are already transferred to a client for visualization, then the images might immediately be processed on the client to save the time to process and move the data from the web server to the client. If a client does not have sufficient processing power to meet performance requirements, then the images may be processed on a web server and then transferred to a client. The choice must be made during development by making assumptions about the server-client transfer rates and relative comparison of server-client processing power. The current implementation of WIPP assumes the classification of computations as illustrated in Fig. 6.3.

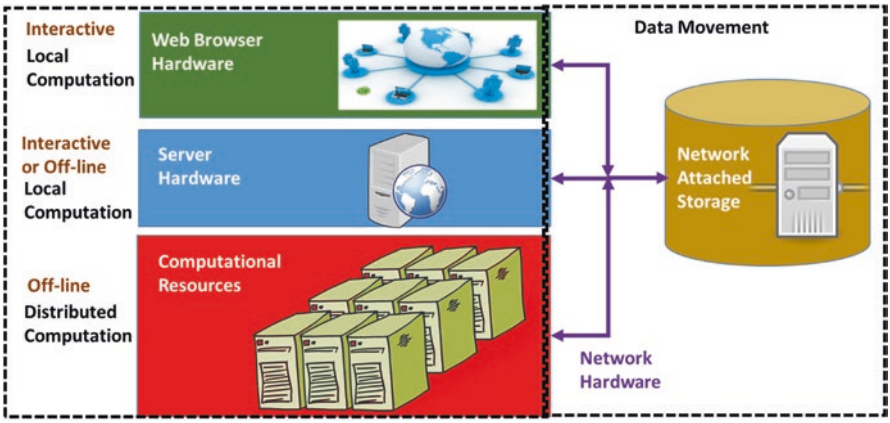

Fig. 6.3 Interactive versus off-line and local versus distributed computations in the context of the WIPP client-server system

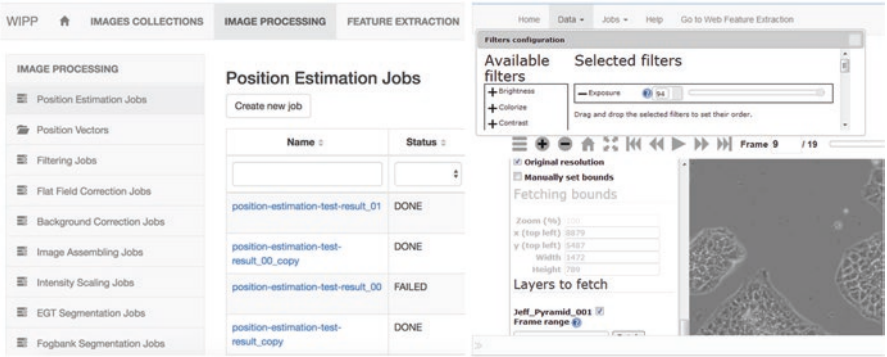

Fig. 6.4 Left – computational jobs executed by a web server on distributed computational resources. Right – image filtering operations executed by a web client on a local client machine

Strategies for off-line vs interactive computations in WIPP

In WIPP, all computations accessible from the Image Processing or Feature Extraction menus (see Fig. 6.4 left) are scheduled by the server to be executed on distributed computational resources. In this case, image processing is applied to the entire image collection and is classified as an off-line computation. On the other hand, computations launched inside of the Deep Zoom viewer (see filtering options in Fig. 6.4 right) are executed on a client computer. Client-side image processing is only applied to images viewed in the browser. For the most client computers that are running web browsers, these image filtering computations are interactive. When images are viewed in the Deep Zoom viewer, the web server processes all requests for image tiles and sends them to a web browser as users are panning and zooming. These computations are handled by the web server and are typically interactive depending on the client-server connectivity.

6.3.2 Estimating and Measuring Execution Time

Definition of execution time

To meet execution time requirements, we need to compare multiple acceleration strategies based on a performance metric. We describe execution time and relative speedup as the two main performance metrics. The execution time is derived from the number of clock cycles per execution divided by the clock rate of a processor. For instance, if a program execution takes one million clock cycles on a processor with 1 MHz clock rate, then the execution time is 1 s. The speedup is derived as a ratio of the execution time and the reference execution time for a given computation. One could improve the speedup by using parallel programming models and more computational resources. We divide approaches to obtaining benchmarks for the two metrics into estimation and measurement categories.

Estimation of execution time
Estimation of the execution time for a given task and its algorithmic implementation is possible by evaluating the number of clock cycles per program execution. The difficulty lies in the mapping of complex program computations into clock cycles and then including other contributions to the overall execution time for the entire end-to-end system. Such evaluations almost always involve many approximations. Nevertheless, according to computational complexity theory, computational problems can be classified into classes based on the estimated orders of the number of clock cycles per program execution. For example, if a program must process n image pixels and the number of clock cycles per pixel is 4, then the total number of clock cycles is 4n. This linear computational complexity as a function of n inputs is denoted as O(n) in big O notation. The big O notation hides constant factors and smaller terms. It is very useful for classifying problem independently of hardware specifications.

Estimation of speedup
Relative speedup measurements might be an alternative estimate of interest. Given the cost of advanced hardware and the amount of time spent writing parallel programs, one may want to predict expected computational speedups as a function of the needed investments of time and money. In the case of WIPP, the speedup for a deployment on a computer cluster or cloud can be predicted using Amdahl's law [2]. The speedup S is defined as a ratio of the execution time on a single machine $T(1)$ over the execution time on P processors being utilized in the cluster $T(P)$ as presented in equation below:

$$S(P) = \frac{T(1)}{T(P)} = \frac{1}{\alpha + \frac{1}{P}(1-\alpha)} \tag{6.1}$$

where α is the nonparallelizable fraction of an algorithm. Amdahl's law assumes that the input data size and the amount of computation are fixed. This is typically the case of uploading an image collection from one experiment and processing it on a deployed instance of WIPP.

It is possible to redesign algorithms in WIPP as computational hardware becomes faster. In this case, Gustafson's law [3] can be used instead of Amdahl's law to fully exploit the improving computing power over an increasing input data size in a fixed execution time. Gustafson's law is shown below with the same notation as above:

$$S(P) = \alpha + P(1-\alpha) \tag{6.2}$$

Measurements of execution time
Finally, one can collect actual execution time benchmarks on a specific hardware and software configuration. There are three types of time measurements:

1. Wall clock time: the observed time elapsed between the start and the end of the program measured by an external clock
2. User CPU time: the total time used by the computer's processor executing just the code of the user's program

3. System CPU time: the total time used by the processor executing kernel code (i.e., the core of an operating system) on behalf of the program

The kernel code is called from a program, for instance, when read or write operations are performed (also called the system calls). In a case of parallel computations, wall clock time is usually less than user CPU time because the program is run concurrently with other programs and must also be waiting for disk, network, or other devices.

Practical notes about execution time and speed-up metrics
Absolute execution time measurements are typically obtained as an average over a set of repeated runs, which are needed because of varying background processes running concurrently with the measured program. The disadvantage of absolute measurements is that they are hard to use for predicting execution times with different hardware and software configurations. If these configurations remain constant and are replicated across the multiple computational nodes of a cluster, then one can collect speedup benchmarks and use them for predictions. These benchmarks capture an execution time as a function of the number of nodes. They provide better understanding of computational scalability and are useful for trade-off decisions between the shortest execution time and the minimum cost of computational nodes. If both software and hardware configurations across all computational nodes are not the same (e.g., heterogeneous computer cloud), then the speedup benchmarks correspond to the ratio of the worst-case execution time of the fastest sequential algorithm on one of the nodes to the worst-case execution time of the parallel algorithm on all the computational nodes.

Remark
While we have focused on execution time, we omitted the discussion about the amount of RAM required by a program (or space complexity). This type of analysis is important in the case of big image data and must be considered when choosing a parallel programming models and a hardware architecture. Although it is recommended to collect memory benchmarks while writing a program, the space complexity estimation and RAM consumption measurements were not included in the scope of this book.

6.4 From Commercial Big Data Analytics to Research Big Image Analyses

There is a wealth of knowledge gained from building commercial big data analytic solutions that could be leveraged when designing big microscopy image analytic solutions. To learn from them, one can take the following steps:

- Narrow down big data attributes (4 to 7Vs) in commercial applications to those in microscopy imaging laboratories.
- Extract basic and advanced design considerations.
- Apply the design considerations to the design of big microscopy image analyses.

We describe these steps in the rest of this section.

Microscopy image attributes
For microscopy imaging laboratories, image collections are typically of the order of terabytes with velocity about 100 MB/s, and variety is represented by file formats, imaging instruments, and imaged specimens. Veracity is present in microscopy images due to manually selected microscope settings and many calibration protocols. The key characteristic of images is that they have always spatial grid structure as opposed to unstructured data (ignoring for now image annotations) occurring in many commercial big data sets.

Basic design considerations
As commercial solutions address big data analytics for all big data attributes, basic general design considerations can be observed in all big data solutions.

- Solutions must be modular in terms of hardware and software because data attributes, algorithms, and hardware specifications change all the time and modules must be replaced/upgraded (i.e., survivability).
- Software must utilize hardware to its maximum but also must handle hardware failures (i.e., utilization and profit including redundancy and reliability).
- Solutions should support creating processing workflows (i.e., flexibility via functional reconfiguration).
- Data must have identifiers, immutability, and introspection (i.e., data persistence. The data elements are found using unique identifiers, are stored in perpetuity, and can describe themselves in terms of content and relationships [4]).
- User interfaces to installation and operation aim at "zero installation time" and "zero user interface" (i.e., minimum barrier for users).

Additional design considerations
Beyond the basic consideration, big data analytic solutions must also have:

(a) Data access management and tools for data de-identification for information privacy
(b) Data format standards and tools for format transformations (legacy data)
(c) Data quality and tools for data cleaning
(d) Data reduction and tools for such transformations
(e) Performance verification and the tools for integrity of data and correctness of functionalities
(f) Software and hardware interoperability and the tools for verifying interoperability of replaced components
(g) Data preservation

Depending on application-specific requirements, these considerations should be included in a solution design.

Applying basic design considerations
WIPP has incorporated some of these design considerations. The software consists of modules, such as the Pegasus scientific workflow for integrating algorithms and HTCondor for utilizing multiple computers. Each image collection, intermediate data product, or computational job is associated with a unique identifier. Once an

input collection is locked before computation, it becomes immutable. A simple query to a database provides information about any collection or executed job. To meet the "zero installation time," a Docker container is used for packaging and deploying the software (and Docker swarm on multiple machines). The "zero user interface" requires more inputs from a community of users and has been implemented so far for the traditional mouse and keyboard devices. Future modifications to WIPP will also address specific additional design considerations.

Incorporating a spectrum of application-specific hardware and software considerations is not trivial. We selected three parts of big image analytic solutions that are of concern to users, algorithmic contributors, and web system developers:

1. *Human interface*: how to interface human inputs and interactions with the output of a big image data solution
2. *Storage and data structures for big images*: how to organize and store large volumes of complex image data
3. *Parallel computations*: how to break image computations into task- and data-parallel components

We will focus in the rest of this chapter on these key parts of a client-server solution for processing big images.

6.5 Human Interfaces for Big Image Data Analytics

Spectrum of User Interfaces

We start with the human interface to big data solutions because humans are the most important part of any scientific discovery. Users come with different levels of IT knowledge and experience with software tools. They also pursue multiple goals by executing a sequence of computations. Depending on the user's knowledge, experience background, and goals, user interface (UI) requirements for big data solutions might include:

(a) Predefined menus and buttons for configuring and executing computations
(b) Scripting and plugin templates for automating computations
(c) Application programming interfaces for integrating new functionality
(d) Application programming interfaces for replacing or adding modules to the entire system (i.e., image processing module, feature extraction module, or machine learning module)

The above UIs can also be classified as:

1. Graphical user interfaces (GUIs)
2. Command-line interfaces (CLI)
3. Application programming interfaces (APIs)

The large variability in user interface requirements implies that a big data solution cannot just have one type of interface for all users.

User interfaces in client-server systems
In client-server systems, the user interfaces (UIs) are on both client and browser sides. We will focus only on a client-side GUI consisting of predefined menus and buttons for configuring and executing computations. The client-side GUIs can be customized by researchers who are knowledgeable about HTML5, CSS, and JavaScript. These UIs are of interest since the reader is assumed to be interested with the easy-to-use aspects of web systems like WIPP. We also provide an example of the GUI design process for the web statistical modeling tool.

6.5.1 Focus on Client-Side Graphical User Interfaces

GUI elements
The objective of GUI design is to present interface elements that are easy to use, access, and understand to facilitate the above activities. The interface elements can be classified as follows[5]:

1. Input controls (e.g., execution launch via buttons; selection via radio buttons, checkboxes, drop-down lists).
2. Navigational components (e.g., page navigation via breadcrumb or pagination, search via search field, sequence navigation via slider or icons).
3. Informational components (e.g., short description via tooltips or modal windows, status of computation via progress bar and icons, warning and error reports via notifications or message boxes).
4. Containers (e.g., toggle between hiding and showing multiple functionalities or large amount of content: via accordion in JavaScript, encapsulating image and processing functionality via JFrame using Java Swing library).

To meet the objectives of GUI design, these elements must be integrated following concepts from *interaction design* focused on interactive digital products and services, *visual design* concentrated on print or electronic forms of visual information, and *information architecture* concerned with organizing and labeling online sites and software to support usability and findability.

GUI design
GUI design anticipates user intentions. For WIPP, the client-side GUI assumes that users intend to:

(a) Uploading and downloading data
(b) Searching for image collections and other data types
(c) Selecting and configuring computations to launch on a server
(d) Browsing results of computations
(e) Viewing big images
(f) Selecting and configuring computations to launch on a client while viewing big images

[5] https://www.usability.gov/what-and-why/user-interface-design.html

With a list of anticipated use cases, best practices for designing GUI can be put in place to address simplicity, consistency, use of color and texture, layout, and typography to assure legibility and readability.

6.5.2 *Example of GUI Design for web Statistical Modeling Tool*

GUI design for web statistical modeling tool
Let us consider a GUI design for the web statistical modeling tool described in Chap. 2, Sect. 2.4. A user wants to derive a statistical probability distribution function (PDF) model from cell colonies that have been segmented from a sequence of gigapixel images and described by a set of cell colony measurements (features). The user interface should allow the following actions:

1. Selecting an imaging channel
2. Selecting a colony feature
3. Defining number of histogram bins
4. Filtering data considered for modeling by their spatial location
5. Filtering data considered for modeling by their feature value
6. Computing and showing statistics of selected and filtered data
7. Suggesting PDF model type
8. Computing and showing PDF model parameters
9. Saving the final histogram with traceable hyperlinks for each cell colony to its persistent source

In this example, the functionality is divided into parameter selection (1–3), spatial filtering (4), feature filtering (5), statistical modeling (6–8), and publication (9). The GUI design for parameter selection is implemented using input controls, such as drop-down menus, an edit box, and a spinner. The rest of the functionality is encapsulated in an accordion type of a container (see Fig. 6.5). Within the accordion, all statistical modeling and publication functions are launched using input controls, such as buttons with icons, which are followed by informational components, such as message boxes, images, and tooltips. In contrast, all filtering functions are using either slider bars for feature values or an image viewer for cell colony locations shown with color-coded markers based on a feature value.

General challenges in GUI design
Perhaps, one of the most general challenges in GUI design is the limited size of device displays. Depending on the stage of user's activities (selecting parameters→filtering→statistical modeling), different information is more (or less) important to users for making their decisions. Thus, the GUI could reallocate the use of display size depending on the activity which was achieved by an accordion element (hiding and showing input controls) in the example above. This concept has also been implemented in the design of integrated development environments (IDEs) where

Fig. 6.5 GUI of web statistical modeling tool with the accordion type of a container (left) and a canvas for display (right)

Fig. 6.6 Filtering challenges in the web statistical modeling tool in terms of display size

switching, for example, between programming, debugging, and searching activities, triggers new layouts.

Specific challenges in GUI design for big image data

Specific challenges in big image data arise when showing information for spatial and feature filtering since there is no display size that could accommodate gigapixel images and thousands of feature histograms with sliders (see Fig. 6.6). For spatial filtering, one must adopt multi-resolution representations of gigapixel images to enable pan and zoom. For feature filtering, one can use a scroll bar to view feature histograms beyond those that fit on a finite display.

6.5.3 Summary

In summary, GUI design for big image data must address both general and specific design challenges and incorporate all basic design principles. The interactivity aspects of GUI design must be understood in the context of the requested computations to be completed. For example, image thresholding computations requested over a TB-sized image might take more than a mouse click, while the same computation over a MB-sized image could be completed within the interactive time definition. This implies that a GUI design for visual optimization of a threshold parameter over a MB-sized image would have an interactive interface (click and render result). In comparison, a GUI design for the same computation over a TB-sized image would have an interface that consists of unique identifiers to check the status of the computation completion (i.e., status message or hour glass per unique identifier) and an interface to retrieve its results. Finally, a GUI design can become very complex software (see the source code for Web Deep Zoom Toolkit[6]), and therefore modularity of the code should also be considered.

6.6 Storage and Data Structure for Big Images

We described the pyramid representation as a data structure for big images in Chap. 4 (Representation of Large Images). We mentioned heterogeneity of image pyramids in terms of their file formats. Here we provide a broader perspective on storage layouts for big images and their data structures in RAM.

6.6.1 Storage for Big Images

Types of storage layout
We are concerned with storing a very large image on a disk, reading and writing its image content efficiently, and preserving all information accompanying all images acquired by a microscope and all information generated during image processing. To achieve maximum performance with big images, one must understand multiple types of storage layouts and their impact on the end-to-end execution time. Figure 6.7 illustrates (1) disk, (2) file storage, (3) image pixel, and (4) pixel byte layout options. We elaborate next on each storage layout option.

Disk storage layout
Big images can be stored on disk(s) in:

1. A single file
2. Multiple files stored as a set of folders on a file system (i.e., the multi-resolution pyramid representation)
3. A database

[6] https://github.com/usnistgov/WebDeepZoomToolkit

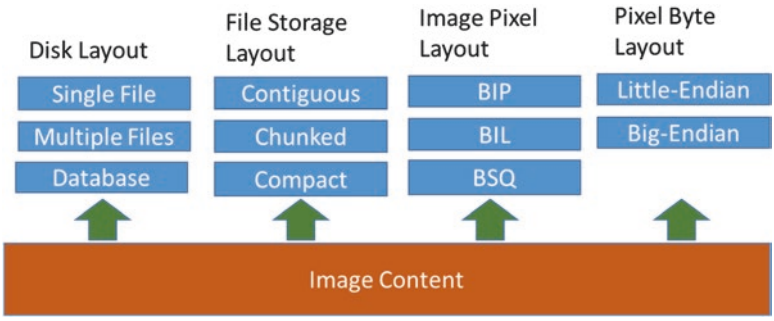

Fig. 6.7 Options of storage layouts for image content. The image pixel layout abbreviations stand for Band interleaved by pixel (BIP), Band interleaved by line (BIL), and Band sequential (BSQ)

In any big image experiment, it is very likely that at least two of these storage types will be used. The reason lies with the fact that to the best of our knowledge, all microscopes store acquired FOVs in a single file. If the acquired files are preprocessed for visualization on the web, then they are likely stored on a file system or in a container file format. If the acquired files are analyzed, then they are likely stored in a database with the image measurements. It is also very likely that an acquired image will change not only its storage type but also its file format at some time point. This is due to (1) a large spectrum of single file formats for storing images acquired by a variety of microscopes and (2) the multiplicity of visualization and analytical goals that an image might support during discovery. Preserving all information during these storage and format conversions is very important for traceability and reproducibility of imaging experiments.

Single file storage

The most commonly used single file formats in microscopy are Adobe TIFF (Tagged Image File Format) and HDF5 (hierarchical definition file, version 5). Both formats have open-source implementations of libraries for reading and writing files. The TIFF specification is described in the 6.0 specification [5], and its implementation in C programming language is available online.[7] The HDF5 specification and its implementation in C programming language are maintained by the HDF group.[8]

TIFF (single file storage)

For files in TIFF format, the topic of preserving information in a single file has been addressed by converting TIFF files into a standard Open Microscopy Environment (OME) representation (see Chap. 5, Loading Images Using OME Bio-Formats Library). The TIFF files acquired by a microscope are stored in OME-TIFF[9] format using the Bio-formats library[10] developed under the umbrella of the Open Microscopy Environment consortium. The topic of TIFF storage size has been

[7] http://libtiff.org/

[8] https://support.hdfgroup.org/products/

[9] http://www.openmicroscopy.org/site/support/ome-model/ome-tiff/specification.html

[10] http://www.openmicroscopy.org/site/support/bio-formats5.4/

recognized as an issue in the past because the format has a limit on the stored file size of less than 4 Gibibyte or GiB (4.294967296×10^9 bytes). This limit is due to the 32-bit offset in the TIFF file formats.[11] To overcome this limit of TIFF file formats, the BigTIFF file format specification with 64-bit offsets was introduced in 2007, and the TIFF library called LibTIFF was upgraded as of its version 4.0.[12] Similar issues arose in the past with the number of bits per pixel (BPP). While the original TIFF format supported only 2, 8, and 16 BPP, the format has been extended to support 32 BPP based on the need of many geographical mapping agencies using geospatial information systems (GIS). Although the LibTIFF library (version 4.0) has the support for 64-bit offsets and 32 BPP, many image processing packages still contain older versions of LibTIFF and hence have only support for the 4 GiB limited TIFF format and the pixel depth up to 16 BPP.

HDF5 (single file storage)
For files in HDF5 format, information associated with images is preserved by adding the OME metadata to the HDF5 container (e.g., by using the Bio-Formats library). Due to the popularity of HDF5 for storing very large files, research and commercial communities have built several programming and scripting interfaces to HDF, for instance, API from Java, Python, R, Fortran, lmaris,[13] or MATLAB.[14] The HDF5 format comes with a data model and a library as described in [6]. In terms of HDF file storage size, there is no limit.[15] However, there is a limit of 32 dimension dataspaces (number of bands or channels). In terms of reading and writing speed, HDF5 format contains serial and parallel HDF5 code, and the choice is selected when building HDF5. The key feature of parallel HDF5 is that certain groups of functions must be called collectively if the data value on a target storage node will be modified. Parallel HDF5 can be optimized to maximize the access to image content. The optimization of parallel HDF5 includes setting HDF5 parameters (e.g., chunk size and dimensions), MPI I/O parameters (e.g., the block size and the number of target nodes to be used for collective buffering file access), and parallel file system parameters (e.g., the size of the striping unit and the number of I/O devices to stripe across).

Storage in a set of folders on a file system
In the case of a single file storage, some file formats can store multiple images and/or image chunks. In contrast, storage of multiple images in a set of well-organized folders might be sometimes more efficient. For example, image pyramid representation can be stored in a set of folders labeled by the pyramid level as illustrated in Fig. 6.8. When accessing image content, the file path can be automatically generated according to the zoom level that corresponds to the folder name. For instance, the folder name 0 refers to low magnification images, and the name 13 refers to high magnification images in Fig. 6.8. The files are consistently named per their grid

[11] https://en.wikipedia.org/wiki/TIFF
[12] http://www.simplesystems.org/libtiff/
[13] http://www.bitplane.com/imaris
[14] https://www.mathworks.com/
[15] https://support.hdfgroup.org/HDF5/faq/limits.html

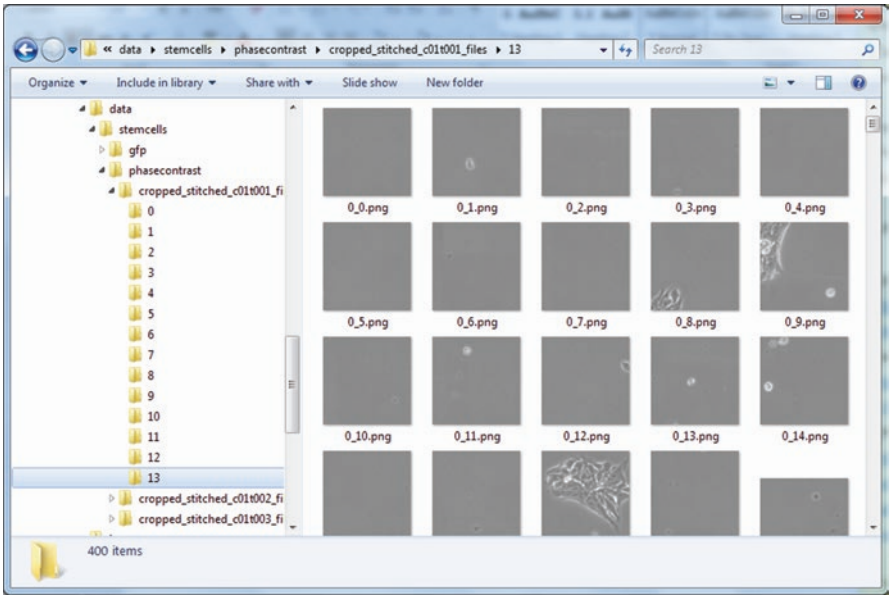

Fig. 6.8 Example of pyramid representation storage in a set of folders on a file system

position (e.g., 0_3.png corresponds to 0th row index and 3rd column index) and hence can be retrieved from the file system efficiently. In this case, we've used the file png format because it is supported by the majority of web browsers.[16]

Database storage

The most common solution for storing big images is to use a file system for storing image chunks and a database for storing indices pointing to the image chunks. Other solutions depend on the size of image chunks, the read/write ratio, and the disk/memory ratio relative to the read/write ratio. If both, large image chunks and their indices, are stored within a database, then the database introduces an overhead by compacting the index look-up table (requires movement of data), accessing data from a fragmented index table (requires inline storage of image chunks in a base table row), and creating many replicates (e.g., log files and database entries, redundancy of immutable data in Hadoop Distributed File System).

File storage layout

The storage layout defines how the pixel values are physically stored on disk. We use the TIFF and HDF5 file formats to explain multiple file storage layouts. For example, HDF5 supports three storage layouts:

1. Contiguous
2. Chunked
3. Compact[17]

[16] https://en.wikipedia.org/wiki/Comparison_of_web_browsers

[17] https://support.hdfgroup.org/HDF5/Tutor/layout.html

Contiguous refers to the pixel values stored in one contiguous block of the HDF5 file. In comparison, chunked denotes pixels being stored in equal-sized contiguous blocks (chunks of a predefined size) together with a chunk index to keep track of their association with a dataset. Compact storage layout was designed only for small datasets that can be stored in the HDF5 header of the dataset.

In comparison, TIFF supports contiguous and chunked storage layouts. The key difference is with the chunked layout type where the TIFF tags (RowsPerStrip, StripOffsets, and StripByteCounts) encode separate image strips rather than rectangular image blocks. Otherwise, compressed or uncompressed image data can be stored almost anywhere within a TIFF file. The chunked storage layout is the most important layout type for efficient big image input/output (I/O) operations.

Image pixel layout

Image pixel layout can affect the speed of I/O operations. There are three main image pixel layouts:

1. Band interleaved by pixel (BIP)
2. Band interleaved by line (BIL)
3. Band sequential (BSQ)

These image pixel layouts are schemes for storing multispectral 2D images and their pixel values in a file or in a memory. The three layouts are illustrated in Figs. 6.9, 6.10, and 6.11.

The choice of an image pixel layout depends on the expected image manipulations. For example, the BIP layout is optimal for computing a weighted sum of spectral values at each pixel (spectral analysis). The BSQ scheme is optimal for performing spatial filtering on a single spectral band (spatial analysis). The BIL layout can be viewed as a compromise format for easy access to both spatial and spectral information. All image pixel layouts can accommodate any number of spectral bands.

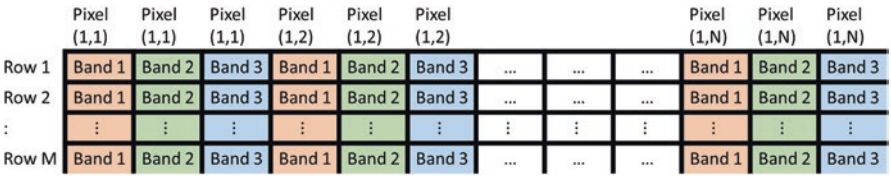

Fig. 6.9 Band interleaved by pixel (BIP) layout of image pixels

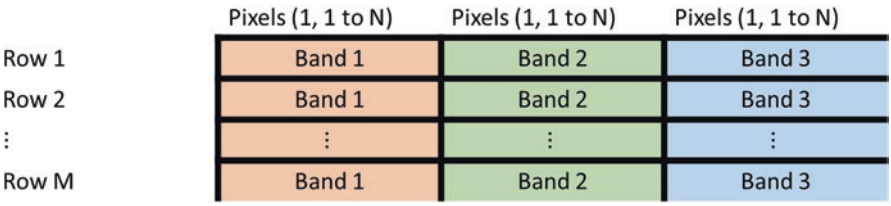

Fig. 6.10 Band interleaved by line (BIL) layout of image pixels

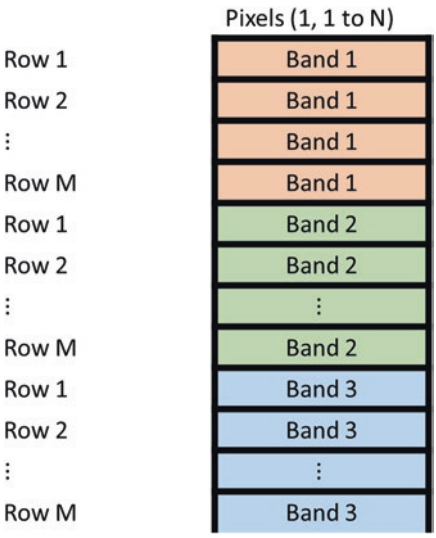

Fig. 6.11 Band sequential (BSQ) layout of image pixels

Pixel byte layout
Each pixel is associated with a value represented by 2, 8, 16, 32, or 64 bits per pixels. During network transmission, byte order is important for values represented by more than 8 bits (1 byte = 8 bits). There are two sequential byte orders: little-endian (least significant bits first) and big-endian (most significant bits first). For example, the number 5 would be binary-encoded using 16 BPP as 0000 0101 0000 0000 using big-endian and 0000 0000 0000 0101 as little-endian. The two sequential orders have implications on storage and are encoded in the file's metadata section.

6.6.2 Data Structures for Big Images

Data structures are the representations of images in RAM. Due to the finite size of RAM, we categorize big image data structures based on the ratio of RAM size to image size. We will briefly mention images with more than two dimensions (denoted as 2D+) and their pyramid representations. WIPP supports 2D+ image collections of videos and spectral sequences with gigapixel 2D images (i.e., XY + time or $XY + \lambda$).

Images smaller than available RAM
If an image can fit into RAM, then it can be represented as a multidimensional array of values. Because many image operations require accessing all the pixels using programming language looping constructs, a one-dimensional array is an efficient representation for high-dimensional images. In the body of the loop, subsequent pixels are accessed by incrementing an index, which is more efficient than calculating the position of the next pixel from the row, column, and band values. For

example, if the image is in the BIP layout, then a pixel value at the location [row, column] is computed as

$$\text{pixel index at}\left[\text{row,column}\right] = \left(\text{number of columns} \times \text{row} + \text{column}\right) \times \text{number of bands} + \text{band} \tag{6.3}$$

Images larger than available RAM
If an image cannot fit in the available RAM, then it must be represented as a combination of multidimensional arrays with the indices defining location in relative to the current big image sub-array. This representation can be viewed as a coordinate transformation from a 2D Cartesian system (i.e., big image rows and columns) to a 4D or 5D Cartesian coordinate system that enables access to image subareas that fit in RAM. The 4D Cartesian coordinate system consists of a set of small images (chunks) with a corresponding position vector for their coordinates in the big image. In other words, the 4D system contains one set of 2D coordinates giving the position of a small image tile in a regular or irregular grid and the other set of 2D coordinates referring to a pixel position inside of a small image tile. The 5D Cartesian coordinate system consists of a multi-resolution pyramid of small images (tiles). It can be viewed as a location in a stack of 4D Cartesian coordinate systems (or a 5D pyramid) where the fifth dimension is the resolution. This representation has been proven to be very efficient for big 2D images.

2D+ images
In practice, microscopy images are not more than two-dimensional. However, microscopy experiments over a large FOV can generate time-lapse images (e.g., *XY* + time videos from phase contrast microscopes), high-dimensional spectral images (e.g., $XY + \lambda$ data from coherent anti-Stokes Raman microscope or scanning electron microscope with energy-dispersive X-ray spectrometry), confocal z-stack images (e.g., XYZ data from confocal laser scanning microscope), or a combination of time-lapse, spectral, and z-stack images (e.g., $XYZ + \lambda$ data from confocal z-stack images with more than three fluorescent channels or *XYZ* + time video from confocal laser scanning microscope).

Pyramids for 2D+ images
We can view these high-dimensional terapixel images as a set of 2D cross-sectional images. If we decompose 3D+ images into 2D cross sections, then we can use a pyramid representation suitable for big 2D images. These 2D cross sections can be represented as an ordered set of multi-resolution pyramids using one pyramid per 2D cross section with the order defined by the third dimension. To enable fast rendering and processing, one may have to generate three sets of pyramids that correspond to the three orthogonal 2D cross sections per one large 3D volume (image). For example, for a 3D image with *XY* + time dimensions, one would generate sets of pyramids for {*XY*}, {*X* + time}, and {*Y* + time}. This representation works well for *XY* + time or $XY + \lambda$ images where oblique views are not meaningful for visual inspection. However, this representation might be limiting for 3D images with *XYZ* dimension images because the oblique views are important for data explorations.

6.6.3 Summary

In the design of a big data analytic solution, there are many decisions about big image storage and representation that determine image content access efficiency. It is possible to optimize the design for a given set of big image analyses with a well-defined pattern of accessing image content. Given hardware specifications (RAM, disk, and bus), optimal parameters can be determined in terms of pixel byte, image, file, and disk storage layouts as well as data structures. In practice, it can be difficult to predict image content access patterns and anticipate the hardware used. While this unpredictability can explain suboptimal image analytic software performance, it also highlights the importance of considering application requirements and usage patterns when making software design decisions.

6.7 Parallel Computations Over Big Image Data

Software development for big data must address three problems:

1. Algorithmic design to automate processing
2. Integration of heterogeneous algorithms into software systems to leverage existing software investments
3. Algorithmic implementations that integrate software and hardware

The last problem of integrating software with distributed hardware resources is the topic of parallel computing research. We provide a high-level classification of parallel programming models and briefly describe each model.

Parallel computing
The basic premise for accelerating big data image computations is that (a) the images can be divided into smaller tiles and (b) the computations can be divided into smaller functional tasks that are then applied in parallel to the smaller image tiles (image data and functional decompositions). Parallel execution accelerates calculations by utilizing multiple computational resources but comes with the cost of additional hardware resources and of writing the software specific to a hardware architecture.

Classification of parallel programming models
The several commonly used parallel programming models abstract hardware and memory architectures and provide different programming approaches. Parallel programming models[18] that are derived from computer architectures include:

1. Shared memory model (without or with threads)
2. Distributed memory model with Message Passing Interface (MPI)
3. Partitioned Global Address Space (PGAS) model with data parallel decomposition
4. Hybrids of the above

[18] https://computing.llnl.gov/tutorials/parallel_comp/#Whatis

Parallel programming models can be divided into two broad categories based on their structural granularity of their parallel programs:

1. High-level granularity
2. Algorithmic-level granularity

These categories can be further subdivided. Those with high-level granularity have been divided into:

1. Single program multiple data (SPMD) model
2. Multiple program multiple data (MPMD) model

while those with algorithmic-level granularity[19] can be further classified as:

1. Data parallel model
2. Master-agent model
3. Task graph model
4. Task pool model
5. Producer-consumer model
6. Hybrid model

These categories might be overwhelmingly complex for a WIPP user and somewhat complex even for a developer of WIPP algorithms since they require basic understanding of hardware and software.

In this chapter, our approach is to introduce the WIPP developers of algorithms to parallel programming models at the algorithmic-level granularity and incorporate their knowledge about the hardware used for running WIPP in their algorithmic design. We assume that a reader is familiar with a hardware architecture running WIPP which typically includes RAM, CPUs, communication buses for exchanging data, and pluggable graphics processing units (GPUs). Next, we will briefly describe each of the algorithmic-level models applicable to such hardware architectures in the context of image processing.

6.7.1 Data Parallel Model

This model is based on dividing images into smaller regions and applying the same processing to each region on a separate hardware resource (i.e., a computational node). The most difficult aspect is determining the image partition strategy that will colocate each computation with its needed data [7]. To illustrate the difficulty, we list a few spatial image computations in Table 6.1 as examples motivating different partition strategies. For instance, if computations operate on a single pixel (e.g., thresholding that has no spatial overlap with the computation of a neighboring pixel), then image partition can be based on either physical location of a pixel in a file or logical location of a pixel in an image. However, if computations operate on

[19] https://www.tutorialspoint.com/parallel_algorithm/parallel_algorithm_models.htm

Table 6.1 Subdivision of spatial image computations and its relevance to image partition

Types of spatial computations: examples	Input image region	Overlap type	Desired image partition	Input to logical partition
Pixel-based: Thresholding	Fixed size	No overlap	Physical or logical without overlap	None
Kernel-based: Convolution	Fixed size	With overlap	Logical with overlap	Kernel area size
Segment-based: Feature extraction	Variable size	No overlap	Logical without overlap	Mask
Bounding box based: Background correction	Variable size	With overlap	Logical with overlap	Bounding boxes

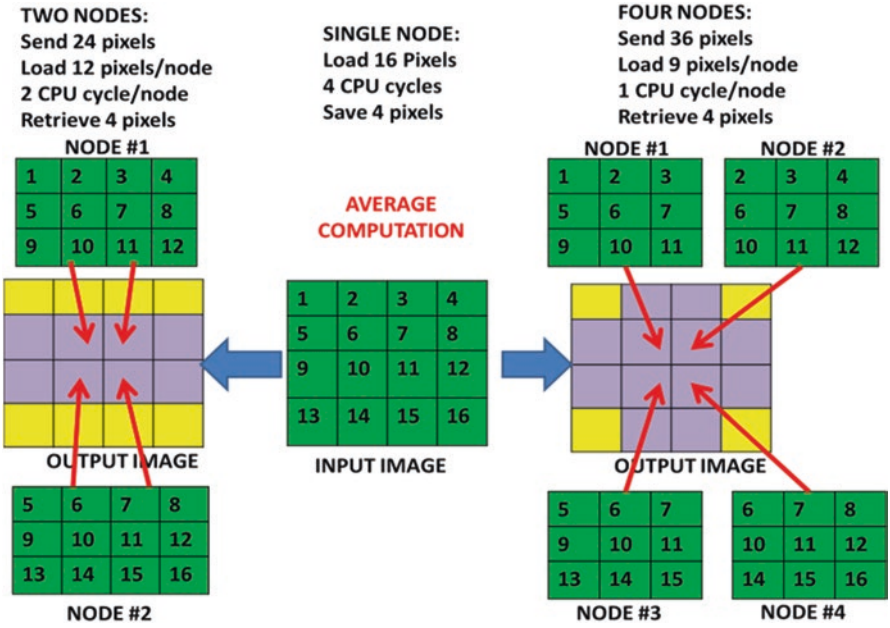

Fig. 6.12 Examples of image partitioning for spatial averaging over 3 × 3 pixels for one (middle), two (left), and four (right) distributed computational nodes

a pixel neighborhood (e.g., kernel-based), then the image partition strategy should be based on logical location of pixels.

Example

As a specific example, Fig. 6.12 numerically evaluates the advantages of logical partitioning for the case of spatial averaging over 3 × 3 pixels with overlapping four image regions. The computation is illustrated for a number of computational nodes N equal to one (middle), two (left), and four (right). The comparison across an increasing number of nodes shows how the decreasing numbers of CPU cycles and the pixel load operations to RAM are counterweighted by an increasing number of communications (send and retrieve operations).

Practical notes
The partitioning can be performed by horizontal and vertical image cuts. To avoid any exchange of pixels between nodes during runtime, the cuts are made in such a way that there is redundancy while spraying pixels across computational nodes. The goal is (a) to avoid the communication overhead between nodes and the storage area network (SAN) or network-attached storage (NAS) where the data is stored and (b) to mitigate the impact of node failure rates of large computer clusters.

6.7.2 Master-Agent Model

The master-agent model is based on introducing a hierarchy of computational nodes to divide the work into (a) "generating" computational jobs and (b) executing the jobs. Figure 6.13 illustrates the master-agent model. The word "generate" refers to decomposing a workflow to individual tasks, collecting data needed for each task in a case of distributed memory, and assigning and transmitting data to agents. One or more master processes manage the computational jobs by delegating them to agent processes so that they are executed in the shortest time, a difficult process known as "load balancing." The load-balancing strategy can consider more than the agent utilization. For instance, other potential factors to consider are data throughput times, agent reliability, past agent execution times, and the patterns of incoming computational tasks.

Job scheduler to execute load balancing
The software that implements load balancing is called a job scheduler or a distributed resource manager. Job scheduling is concerned with assigning computational jobs to computational nodes which is distinct from operating system process scheduling concerned with assigning running processes to CPUs. The input to a job scheduler is a job queue which contains job information. Job schedulers are frequently embedded in scientific workflow systems, such as the Pegasus workflow system used by WIPP. The workflow for a sequence of computations also defines task dependencies that are utilized in master-agent models and in job schedulers.

Hadoop example of a master-agent model
An example of the master-agent model is the Apache Hadoop framework,[20] which is designed for storing data and running big data computations on computer clusters and clouds consisting of commodity hardware. Data storage is supported by the Hadoop Distributed Filesystem (HDFS) which uses the master-agent model. In this model, one cluster node (labeled as NameNode) manages file system operations, and a set of agents (labeled as DataNodes) manages data storage on their individual cluster nodes. When NameNode sprays data blocks across DataNodes, the blocks are replicated multiple times. The defaults are 64 MB blocks with two replicates. If a DataNodes fails, then the NameNode finds the replicates elsewhere in the cluster,

[20] https://hadoop.apache.org/docs/r2.7.1/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Mapper

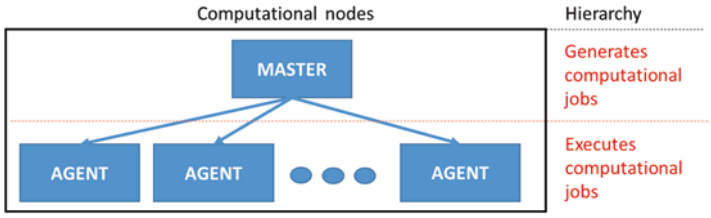

Fig. 6.13 Master-agent model and its hierarchy of computational nodes

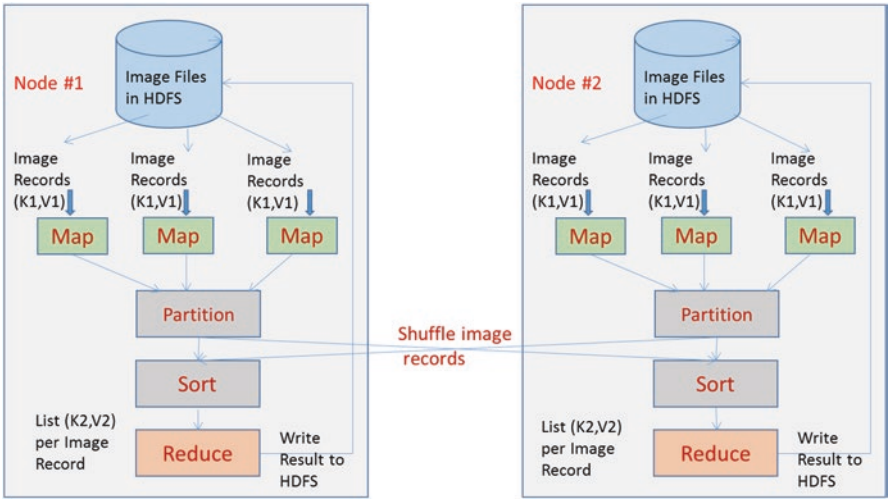

Fig. 6.14 Simple illustration of MapReduce programming paradigm utilizing two computational nodes

allowing the computation to be restarted on a different computational node while the operator replaces the failed hardware.

Map-reduce implementation

Several WIPP image processing algorithms have been evaluated using Hadoop implementations [8]. The computation with Hadoop is based on the MapReduce programming paradigm [1]. We will explain the MapReduce programming paradigm using a simple two-node cluster to perform an image intensity histogram calculation. We start by assuming that image pixels have already been sprayed across HDFS and each node has a subset of the image pixels, as illustrated in Fig. 6.14.

Map function

The image intensity histogram calculation starts with a Map function that computes frequency count for each intensity value over an allocated set of pixels. This Map function can be viewed as a transformation of pairs from one space to another space; the Map transforms a list of pairs (K1 = pixel location, V1 = intensity) to another list of pairs (K2 = intensity value, V2 = frequency count). Once executed, Hadoop will

exchange groups of entries between the available nodes as illustrated in Fig. 6.14 by "partition" and "sort" operations along with the Hadoop shuffling operation. In our example, Hadoop will create Group 1 that has K2 in [0, 128] and Group 2 that has K2 in [129, 255]). Then, Node #1 sends Group 2 to Node #2 and Node #2 sends Group 1 to Node #1.

Reduce function
A previously written Reduce function merges the entries with the same K2 value and saves the resulting counts in HDFS. After the Reduce function, the image histogram counts can be retrieved from HDFS.

Additional functions
Hadoop allows for more direct control with several classes. A Hadoop Comparator object can be used to specify the grouping during the shuffling operation, a Partitioner object can determine the Reducer node for a set of K2 keys, and a Combiner object can decrease the number of shuffled bytes by determining the local aggregation of the intermediate Map.

Notes
As with any other distributed computing software, users must optimize a number of parameters, such as HDFS block size, replication ratio, Hadoop process RAM allocation, number of Map and Reduce instances, and the encryption method for data transfers. Parameter optimization and the computational decomposition into Map and Reduce tasks are the most significant challenges for using systems like Hadoop. Nevertheless, the MapReduce paradigm has been successful for storing and processing commercial big data.

6.7.3 Task Graph Model

The task graph model describes a computation as a directed task graph formed by a collection of vertices denoting atomic tasks and directed edges describing data movement. An atomic task is a logically discrete section of computation in a program that is executed by a processor. A task graph-based parallel algorithm consists of atomic tasks running on multiple processors or computational nodes. All computations are executed by traversing the task graph by following the directed edges. A graph is a directed acyclic graph (DAG) if there are no paths that start at a vertex, follow a sequence of directed edges, and return to the starting vertex. Figure 6.15 shows an example of DAG with eight atomic tasks T1–T8. Programs that can be represented by a DAG are characterized by useful properties, for instance, a reachability relationship. In Fig. 6.15, we illustrate the reachability relationship with vertex T8 reachable from the vertex T1 (T1 $\leq$ T8) if there is a path from T1 to T8.

Practical use task graph models
Practical applications of task graphs include task scheduling, dataflow programming, and management of software revision history and its versions. This

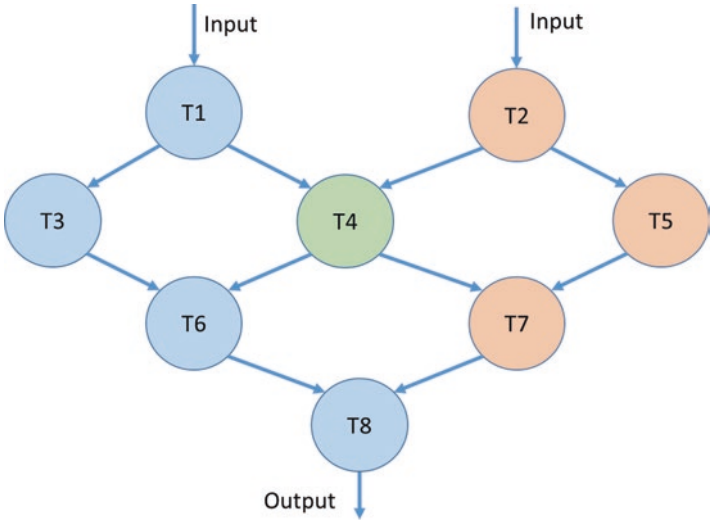

Fig. 6.15 Example of a directed acyclic graph with eight tasks

programming model is recommended for computations where the transfer time of moving data is larger than the total time needed for the number of computations associated with the data tasks. In this case, the task graph structure can be optimized to lower the data movement cost between the tasks. The most difficult aspect of implemented task graphs is the decomposition of computations into atomic tasks, identifying task synchronization and communication dependencies, and the creation of the directed task graph. For example, the color coding of vertices shown in Fig. 6.15 can be used for assigning tasks to three computational resources assuming that all tasks require the same amount of time to complete. The assignment becomes complicated when the computational resources and tasks are heterogeneous in terms of CPU power, RAM, data needs, and computational complexity (i.e., the assignment becomes a load-balancing problem).

6.7.4 Task Pool Model

The task pool model can also be thought of as a type of task graph model. It is based on dynamic assignment of tasks to the computational nodes to balance the load. There is an advantage to creating a pool of tasks when the task completion time is unpredictable or varies significantly. The model consists of the implementations of a master and agent processes. The master generates and holds a pool of tasks, sends tasks to agents upon request, and collects the results. The agents request and receive tasks from the master, execute the tasks, and return the results to master. If the pool of tasks is generated dynamically, then a method of detecting termination is required

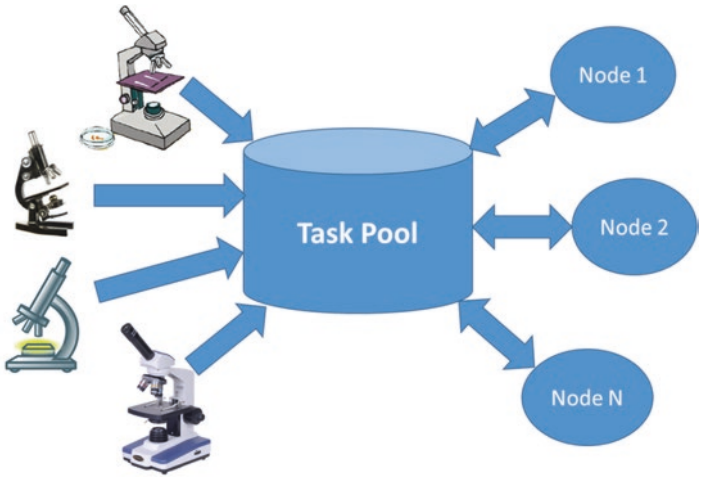

Fig. 6.16 Task pool model with an unpredictable number of tasks generated by image streams from multiple microscopes

so that all agents stop requesting tasks when the supply is exhausted. The number of tasks executed by each computational resource will depend on its speed of execution because there is no preassignment of tasks.

Practical notes
The task pool model is used when the amount of data associated with each task is small and therefore the communication time overhead in sending tasks and receiving results is smaller than the total time needed for computations. One of the difficult design decisions is choosing the task granularity to optimize relationship between the communication overhead, the computational effort, and the data quantities. For example, if the goal is to index acquired images by their average intensity, then a task pool model is appropriate for processing each incoming image from a set of microscopes. As illustrated in Fig. 6.16, the number of images per time unit depends on microscope acquisition rate and its usage pattern which dynamically determines the tasks generated to compute an average image intensity. Computational nodes request the tasks with the time needed to complete a computation depending on the image size. The task pool model inherently balances the computational load despite varying image sizes.

6.7.5 Producer-Consumer Model

The producer-consumer model (also called the pipeline model) represents a computation as a chain of multiple data producers and consumers in a manner similar to an assembly line. Each computational task in the queue consumes data from the preceding task and produces data for the subsequent task. The queue can be linear or

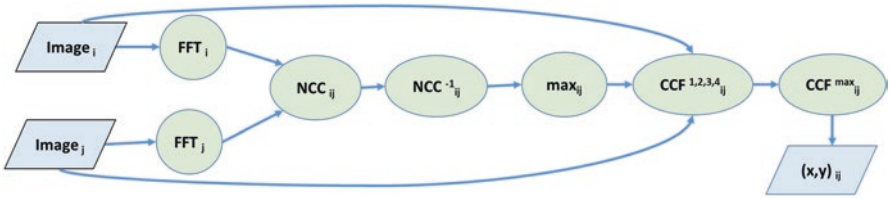

Fig. 6.17 Computation of relative displacements (x, y) of two image tiles (images i and j) using a consumer-producer model

represented by a directed graph. The producer-consumer model is different from a task graph model by overlapping task interactions with computations. Both, producer and consumer, share a common, fixed-size buffer used as a queue. The producer generates data, puts them into the buffer, and starts its task-specific computation again. At the same time, the consumer removes data from their common buffer, one chunk at a time, and starts its task-specific computation. The producer will wait if the buffer is full, and the consumer will wait if the buffer is empty. In other words, every time a producer task generates new data, it triggers the execution of a next consumer task in a queue.

Stitching algorithm

Among several image processing algorithms in WIPP, the tile stitching algorithm has been implemented for multiple architectures using multiple producers and multiple consumers [9]. The tile stitching computation is based on normalized correlation coefficients (NCCs) derived from fast Fourier transforms (FFTs). The image tile displacements (i.e., translation vectors (x, y) for each pair of adjacent tiles) can be computed using multiple data and functional decompositions, as well as CPU and GPU hardware architectures. Figure 6.17 shows the sequence of steps for multi-threaded CPU-only implementation. After each image tile is read into memory and processed by FFT, NCCs are computed. Afterward, 2D inverse FFT of the NCCs is performed ($NCC^{-1}{}_{ij}$), and maximum is found ($\max_{ij}$). The final steps are to compute the cross-correlation factors (CCF) for the north, west, south, and east overlaps, find the maximum CCF, and save the corresponding translation vector (x, y). The key aspect is the assignment and management of threads (one for reading an image tile, one for computing FFT, one for completing NCC, FFT^{-1} and max, and multiple threads for CCF). The difference between simple sequential and pipelined CPU implementations can yield a speedup by almost an order of magnitude and with GPUs even higher [9].

Practical notes

The consumer-producer model has been used for developing general dataflow environments (i.e., computational scenarios where data flow along a processing pipeline). These environments became popular in scientific workflow management systems (e.g., Kepler [10], Taverna [11]) and in commercial frameworks such as the .NET framework (TPL Dataflow Library). A difficulty with the consumer-producer model

is the implementation of multiprocess synchronization. Nevertheless, given the large number of open-source scientific workflows [12] that utilize dataflows, we recommend using one from the existing workflow management system implementations when suitable. For example, a sequence of filtering operations applied to big images can be implemented by using a consumer-producer model to pass the partially filtered image regions in RAM rather than passing them between RAM and disk.

6.7.6 *Hybrid Model*

The hybrid model combines multiple programming models either hierarchically or sequentially to any part of a computational algorithm. Hybrid models are motivated by integrating functional and data decompositions with specific strengths of a hardware architecture.

Example with CPU-GPU hardware
The integration of functional and data decompositions can be achieved by combining CPUs and GPUs at two different levels. At the low level, the integration takes place by putting CPU and GPU on the same die and sharing the on-chip cache and off-chip memory [13]. At the high level, the integration is accomplished by attaching GPUs to a computer with CPUs and orchestrating the split of computations between CPU and GPU units [14]. In the latter case denoted as a single CPU-GPU configuration, the GPUs are typically assigned data parallel work to take advantage of their large number of computational cores, while the CPUs execute sequential code or data transfer management.

Given a CPU-GPU configuration illustrated in Fig. 6.18, the hardware resources consist of several computational nodes with master CPU and multiple GPUs. Following the classification introduced in Sect. 6.7, the utilization of this hardware configuration can benefit from distributed memory model with Message Passing

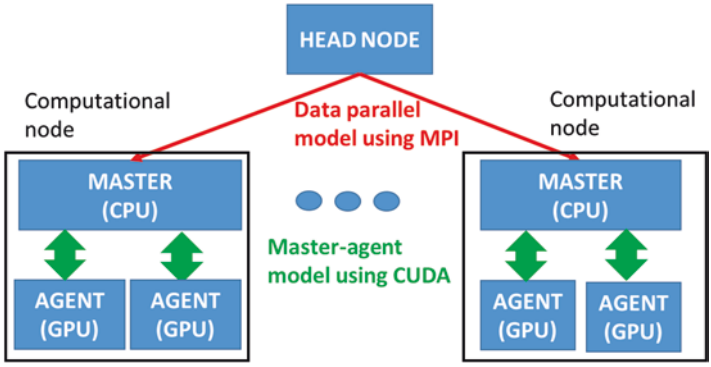

Fig. 6.18 Hybrid parallelization model combining master-agent model using CUDA interface and data parallel model using MPI on a CPU-GPU cluster hardware configuration. CUDA stands for Compute Unified Device Architecture (rarely used in a non-abbreviated form)

Interface (MPI) to run single process, multiple data (SPMD) parallel applications. At the algorithmic-level granularity, one can create a hybrid parallel model consisting of (1) a data parallel model with the MPI protocol (see Chap. 4) for launching tasks on each computational node and (2) a master-agent model for leveraging all GPUs attached to each computational node. In this hybrid programming model, each node receives data from the head node using MPI, exchanges the received data between its local memory and the attached GPUs using GPU-specific interface, and collects computed results. To interface GPUs, one option is to use CUDA®, a parallel computing platform and programming model introduced by NVIDIA.

Practical Notes
In practice, the number of possible hardware configurations is very large, and therefore there is no recipe for creating an optimal hybrid parallel model. Based on the current economics of building computer hardware and providing computer cloud services, developers of algorithms are frequently writing code to run on computer clusters and distributed virtual machines with multi−/many-core machines. In this case, it is beneficial for the developers to be familiar with the parallel programming models described in this section and with the interfaces for direct multi-threaded, shared memory parallelism, such as the Open Multi-Processing (OpenMP[21]) API.

6.7.7 Summary

This entire section focused on parallel computations over big image data and particularly on algorithmic implementations that integrate software and hardware. The algorithmic implementations and approaches followed parallel computing models that were derived from computer architectures and divided based on the structural granularity of their parallel programs. The structural granularity provided a mechanism for classifying parallel programming models into single/multiple program multiple data (SPMD/MPMD) models and a variety of algorithmic-level models (data parallel, master-agent, task graph, task pool, producer-consumer, and hybrid models). The parallel computing premises (data and computation subdivision) and algorithmic-level models were presented at a high level in order to introduce a reader to writing algorithms that leverage hardware.

Forward-looking challenges in algorithmic programming
With the increasing variety of hardware architectures, the parallel programming models are still an open research area. The hardware architectures are changing not only in terms of scale and density (e.g., 5 billion transistors per die, feature sizes close to 10 nm) but also in terms of brand new structures, such as neuromorphic computing and quantum computing. For example, in the early 1990s, scientists began considering a design of brain-like (neuromorphic) computing devices that would dramatically outperform conventional Complementary Metal–Oxide–Semiconductor (CMOS)

[21] https://computing.llnl.gov/tutorials/openMP/#Introduction

based technology. The motivation lies in the fact that the current computational devices have failed to perform many basic tasks that biological systems have mastered, for instance, speech and image recognition. Thus, the next-generation computer design might borrow concepts from biological systems. On the hardware side, computer design might replace CMOS transistors using definite states (0 and 1) with quantum bits using superpositions of states for storing binary digits. These next-generation hardware architectures will require new programming models to utilize them.

References

1. Miner, D., Shook, A.: MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems, 1st edn. O'Reilly Media, Beijing (2012)
2. Hill, M.D., Marty, M.R.: Amdahl's Law in the Multicore Era. University of Wisconsinn, UW CS-TR-2007-1593, Madison (2007)
3. Gustafson, J.L.: Reevaluating Amdahl's law. Commun. ACM. **31**(5), 532–533 (1988)
4. Berman, J.J.: Principles of Big Data. Elsevier/Morgan Kaufmann, Amsterdam (2013)
5. Consortium, "TIFF Specification, Revision 6.0," (1992)
6. Kumar, P., Alameda, J., Bajcsy, P., Folk, M., Markus, M.: Hydroinformatics: Data Integrative Approaches in Computation, Analysis, and Modeling. CRC Press LLC, Boca Raton (2006)
7. Bajcsy, P., Nguyen, P., Vandecreme, A., Brady, M.: Spatial computations over terabyte-sized images on hadoop platforms, In: 2014 IEEE International Conference on Big Data, (2014), pp. 816–824
8. Bajcsy, P., Vandecreme, A., Amelot, J., Nguyen, P., Chalfoun, J., Brady, M.: Terabyte-sized image computations on Hadoop cluster platforms. In: IEEE International Conference on Big Data (2013)
9. Blattner, T., Keyrouz, W., Chalfoun, J., Stivalet, B., Brady, M., Shujia, Z.: A hybrid CPU-GPU system for stitching large scale optical microscopy images. In: Parallel Processing (ICPP), 2014 43rd International Conference on, 2014, pp. 1–9
10. Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., Mock, S.: Kepler: an extensible system for design and execution of scientific workflows. Sci. Stat. Database Manag. 2004 Proc. 16th Int. Conf. **1**, 423–424 (2004)
11. Oinn, T., et al.: Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics. **20**(17), 3045–3054 (2004)
12. Talia, D.: Workflow Systems for Science: concepts and tools. ISRN Softw. Eng. **2013**, 15 (2013)
13. Yang, Y., Xiang, P., Mantor, M., Zhou, H.: CPU-assisted GPGPU on fused CPU-GPU architectures. In: Proceedings – International Symposium on High-Performance Computer Architecture, (2012), pp. 103–114
14. Lee, J., Samadi, M., Park, Y., Mahlke, S.: Transparent CPU-GPU collaboration for data parallel kernels on heterogeneous systems. In: PACT '13 Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, (2013), pp. 245–256