Parallel Computing 40 (2014) 589–610

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Region templates: Data representation and management for high-throughput image analysis

George Teodoro [a,*], Tony Pan [c], Tahsin Kurc [b,d], Jun Kong [c], Lee Cooper [c], Scott Klasky [d], Joel Saltz [b]

[a] Department of Computer Science, University of Brasília, Brasília, DF, Brazil
[b] Biomedical Informatics Department, Stony Brook University, Stony Brook, NY, USA
[c] Biomedical Informatics Department, Emory University, Atlanta, GA, USA
[d] Scientific Data Group, Oak Ridge National Laboratory, Oak Ridge, TN, USA

ARTICLE INFO

Article history:
Received 28 August 2013
Received in revised form 23 September 2014
Accepted 24 September 2014
Available online 2 October 2014

Keywords:
GPGPU
Storage and I/O
Heterogeneous environments
Image analysis
Microscopy imaging

ABSTRACT

We introduce a region template abstraction and framework for the efficient storage, management and processing of common data types in analysis of large datasets of high resolution images on clusters of hybrid computing nodes. The region template abstraction provides a generic container template for common data structures, such as points, arrays, regions, and object sets, within a spatial and temporal bounding box. It allows for different data management strategies and I/O implementations, while providing a homogeneous, unified interface to applications for data storage and retrieval. A region template application is represented as a hierarchical dataflow in which each computing stage may be represented as another dataflow of finer-grain tasks. The execution of the application is coordinated by a runtime system that implements optimizations for hybrid machines, including performance-aware scheduling for maximizing the utilization of computing devices and techniques to reduce the impact of data transfers between CPUs and GPUs. An experimental evaluation on a state-of-the-art hybrid cluster using a microscopy imaging application shows that the abstraction adds negligible overhead (about 3%) and achieves good scalability and high data transfer rates. Optimizations in a high speed disk based storage implementation of the abstraction to support asynchronous data transfers and computation result in an application performance gain of about 1.13×. Finally, a processing rate of 11,730 4K × 4K tiles per minute was achieved for the microscopy imaging application on a cluster with 100 nodes (300 GPUs and 1200 CPU cores). This computation rate enables studies with very large datasets.

1. Introduction

Distributed-memory computing systems consisting of multi-core CPUs and general purpose Graphics Processing Units (GPUs) provide large memory space and processing capacity for scientific computations. Leveraging these hybrid systems, however, is challenging because of multiple memory hierarchies and the different computation characteristics of CPUs and GPUs. Application developers have to deal with mapping and scheduling analysis operations onto multiple computing nodes and, on a node, onto CPU cores and GPUs, while enforcing dependencies between operations. They also need to

* Corresponding author.
E-mail addresses: teodoro@cic.unb.br (G. Teodoro), tony.pan@emory.edu (T. Pan), tahsin.kurc@stonybrook.edu (T. Kurc), jun.kong@emory.edu (J. Kong), lee.cooper@emory.edu (L. Cooper), klasky@ornl.org (S. Klasky), joel.saltz@stonybrookmedicine.edu (J. Saltz).

http://dx.doi.org/10.1016/j.parco.2014.09.003
0167-8191/

implement mechanisms to stage, distribute, and manage large volumes of data and large numbers of data elements across memory/storage hierarchies.

We have developed an analytics framework for on-demand, high throughput processing of very large microscopy image datasets on hybrid computation systems [63,61]. The analytics framework consists of a library of high performance data analysis methods, data structures and methods common in microscopy image analyses, and a middleware system. We reported in an earlier work on methods implemented in the middleware system for scheduling data analysis operations and analysis pipelines on hybrid machines [63,61], In this paper, we investigate efficient data representations and runtime support to minimize data management overheads of common data types consumed and produced in an analysis pipeline.

The primary motivation for this effort is the quantitative characterization of disease morphology at the sub-cellular scale using large numbers of whole slide tissue images (WSIs). This is an important and computationally expensive application in biomedical research. Investigations of tissue morphology using WSI data (also referred to here as microscopy image data) have huge potential to lead to a much more sophisticated understanding of disease subtypes and feature distributions and to enable novel methods for classification of disease state. Biomedical researchers are now able to capture a highly detailed image of a whole slide tissue in a few minutes using state-of-the-art microscopy scanners. These devices are becoming more widely available at lower price points, making it feasible for research groups and organizations to collect large numbers of whole slide tissue images (WSIs) in human and animal studies The Cancer Genome Atlas[1] project, for instance, has more than 40,000 WSIs and counting. We expect that in the next 3–5 years, research groups will be able to collect tens of thousands of digital microscopy images per study, and healthcare institutions will have repositories containing millions of images. Over the past several years, a number of research groups, including our own, have developed and demonstrated a rich set of methods for carrying out quantitative microscopy image analyses and their applications in research [26,51,37,20,45,25,15,16]. Scaling such analyses to large numbers of images (and patients) creates high end computing and big data challenges. A typical analysis of a single image of $10^5 \times 10^5$ pixel resolution involves extraction of millions of micro-anatomic structures and computation of 10–100 features per structure. This process may take several hours on a workstation.

Our earlier work has demonstrated that large numbers of images can be processed rapidly using distributed memory hybrid systems by carefully scheduling analysis operations across and within nodes in the system and that scheduling decisions can be pushed to the middleware layer, relieving the application developer of implementing complex, application-specific scheduling mechanisms. The work presented in this paper introduces a data management abstraction layer, referred to here as the region template framework, for management and staging of data during the execution of an analysis application and shows that the overhead of such an abstraction is small. Our contributions can be summarized as follows:

- A novel region template abstraction to minimize data management overheads of common data types in large scale WSI analysis. The region template provides a generic container template for common data structures, such as points, arrays, regions, and object sets, within a spatial and temporal bounding box. A *data region* object is a storage materialization of the data types and stores the data elements in the region contained by a region template instance. A region template instance may have multiple data regions. The region template abstraction allows for different I/O, storage, and management strategies and implementations, while providing a homogeneous, unified interface to the application developer.
- An efficient runtime middleware to support the definition, materialization, and management of region templates and data regions and execution of analysis pipelines using region templates on distributed-memory hybrid machines. Application operations interact with data regions and region templates to store and retrieve data elements, rather than explicitly handling the management, staging, and distribution of the data elements. This responsibility is pushed to the runtime system. Presently, the runtime system has implementations for memory storage on nodes with multi-core CPUs and GPUs, distributed memory storage, and high bandwidth disk I/O.
- An experimental evaluation of the region template framework on a distributed memory cluster system in which each compute node has 2 6-core CPUs and 3 NVIDIA GPUs. The results demonstrate that this level of abstraction is highly scalable and adds negligible overhead (about 3%).

The rest of the paper is organized as follows. Section 2 provides an overview of the motivating scenario, and the use-case application for integrative microscopy image analysis. The region template framework is described in Section 3. Implementation of global region templates data storage, which are used for inter-machine data exchange is discussed in Section 4. Section 5 presents an experimental performance evaluation of the region template framework. The related work and conclusions are presented in Sections 6 and 7.

2. Background

2.1. Motivation

While our work is primarily motivated by studies that make use of morphological information from tissue specimens, these studies belong to a larger class of scientific applications that analyze and mine low-dimensional, spatio-temporal data

[1] http://cancergenome.nih.gov.

and that have similar data processing patterns. This class of applications includes applications for satellite data processing, subsurface and reservoir characterization, and analysis of astronomy telescope datasets [34,39,50,6,55,14]. Datasets in these applications are characterized by elements defined as points in a multi-dimensional coordinate space with low dimensionality and at multiple time steps. A point is connected primarily to points in its spatial neighborhood. These properties are common in many datasets gathered from sensors and generated from scientific simulations: satellite data in climate studies, seismic surveys and numerical simulations in subsurface characterization, and astronomical data from telescopes.

Rapid processing of data from remote sensors attached to earth orbiting satellites, for example, is necessary in disaster tracking applications as well as for studying changes in vegetation and ocean ecosystems. Accurate prediction of weather patterns using satellite sensor data can assist a researcher to estimate where heavy rain and tornadoes may occur and their paths. In this scenario, it is critical to analyze large volumes of data and corroborate the analysis results using multiple, complementary datasets (e.g., multiple types of satellite imagery). A dataset may define regions of regular, lower resolution grids over the entire continent, while another may contain sensor readings on a higher resolution grid corresponding to a smaller spatial region. The researcher may perform a series of operations to (1) remove anomalous measurements, (2) map data elements in one dataset to another to create regions for full sensor coverage, (3) segment and classify regions with similar surface temperature, (4) perform time-series calculations on land and air conditions, and (5) perform comparisons of conditions over multiple time steps and across spatial regions to look for changing weather patterns.

In subsurface characterization, as another example, scientists carry out simulations of complex numerical models on unstructured, multi-resolution meshes, which represent underground reservoirs being studied, to investigate long term changes in reservoir characteristics and examine what–if scenarios (e.g., for maximizing oil production by different placements of injection and production wells). Data are also obtained via multiple seismic surveys of the target region, albeit at lower spatial resolutions. A researcher may process, combine, and mine simulation and seismic datasets through a series of operations, including (1) selection of regions of interest from a larger spatio-temporal region; (2) mapping data elements from different datasets to a common coordinate system and resolution for analysis; (3) detecting, segmenting, and classifying pockets of subsurface materials (e.g., oil); (4) analyzing changes in segmented objects over time or under different initial conditions; and (5) correlating information obtained from one dataset with information obtained from another dataset (e.g., comparing segmented pockets from simulation datasets with those from seismic datasets to validate simulations).

2.2. Use case: integrative microscopy image analysis

Microscopic examination of biopsied tissue reveals visual morphological information enabling the pathologist to render accurate diagnoses, assess response and guide therapy. Whole slide digital imaging enables this process to be performed using digital images instead of physical slides. The quantitative characterization of microscopy image data involves a process of [16]: (1) correcting for staining and imaging artifacts, (2) detection and extraction of microanatomic objects, such as nuclei and cells, (3) computing and characterizing their morphologic and molecular features, and (4) monitoring and quantifying changes over space and time. In some imaging studies, processing also includes 3-D and/or spatio-temporal reconstruction.

In a typical analysis scenario, nuclei and cells are segmented in each image, and a cytoplasmic space is defined around each nucleus in *the segmentation stage*. Features are calculated for each nucleus to describe its shape and texture in *the feature computation stage*. The properties of the "average" nucleus for each patient is calculated to generate a patient morphology profile. The patient morphology profiles are clustered using a machine-learning algorithm in *the classification stage*. The data are normalized and redundant features are eliminated using a feature selection process. The selected features are processed via statistical and machine learning algorithms to group patients into clusters with cohesive morphological characteristics. The patient clustering results are further processed to search for significant associations with clinical and genomics information in *the correlation stage*. The clusters are checked for differences in patient outcome, associations with the molecular subtypes defined in literature, human descriptions of pathologic criteria, and recognized genetic alterations.

In our current studies the most time consuming stages are the segmentation and feature computation stages. It is highly desirable in research studies to use large datasets in order to obtain robust, statistically significant results, but the scale of an image-based study is often limited by the computational cost of these stages. Modern scanners can generate a whole slide tissue image at up to 120K $\times$ 120K-pixel resolutions. An uncompressed, 4-channel representation of such an image is about 50 GB. Image analysis algorithms segment 10^5 to 10^7 cells and nuclei in each virtual slide of size 10^5 by 10^5 pixels. For each segmented object, 50–100 shape and texture features are computed in the feature computation phase. Fig. 1 presents the computation graph for the segmentation and features computation stages. The graph of operations performed within each of these two stages is detailed. Processing a few hundred large resolution images on a workstation may take days. Distributed memory clusters of multi-core CPUs and modern GPUs can provide the computational capacity and memory space needed for analyses involving large numbers of images.

3. Region templates

3.1. Architecture of region template framework

The main modules of the region template framework are depicted in Fig. 2: the region template data abstraction, the runtime system, and implementations for different data storage, transfer and management mechanisms.

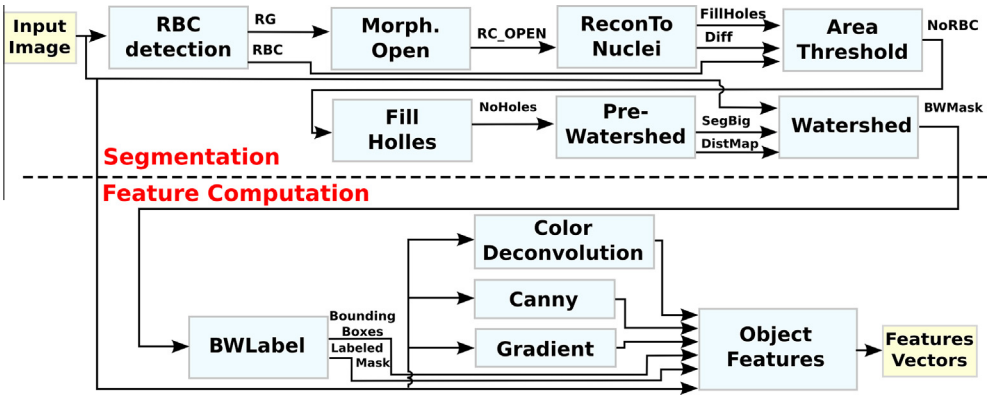


Fig. 1. An example pipeline of the segmentation and feature computation stages. The segmentation phase identifies nuclei and cells in input images and defines a cytoplasmic region around each nucleus. The feature computation stage calculates a vector of 50–100 shape and texture features for each nucleus and cytoplasm. Each stage is internally described as a complex workflow of finer-grain tasks.

Region template applications are represented and executed as dataflow applications. The runtime system implements support for hierarchical dataflows. A hierarchical dataflow allows for an application to be described as a dataflow of coarse-grained components, in which each component may further be implemented as a dataflow of fine-grained tasks. This representation leads to flexibility and improved scheduling performance on hybrid systems, as detailed in Section 3.3. The runtime system instantiates application components/stages and performs multi-level task scheduling to manage the execution of the application components on distributed memory machines and on each computation node. It implements optimizations for efficient execution on hybrid CPU–GPU equipped systems. If there are application-dependent runtime dependencies between application components, it enforces such dependencies for correct application execution.

Data consumed and produced by application components are managed in storage containers provided by the region template data abstraction. Data types supported in this abstraction include data structures commonly used in applications that process data described in low-dimensional spaces (1D, 2D or 3D spaces) with a temporal component. The current implementation supports pixels, points, arrays, matrices, 3D volumes, polygons and surfaces representing segmented objects and regions. The region template data abstraction implements efficient data transport mechanisms for moving data between application components, which may run on different nodes of a distributed memory machine. Instead of writing data through streams as in a typical dataflow application, application components output region template data instances that are consumed by other application components. Dependencies among application components and region template data instances are provided to the runtime system by the application. The runtime system coordinates data transfers while enforcing the dependencies. Data transfers are performed in background to useful computation by I/O threads, which interact with the appropriate implementations of the region template data storage to retrieve/stage data. When transfer of data elements in a region template data instance is completed, appropriate application codes are launched for data processing and scheduled for execution based on scheduling policies implemented in the runtime system.

3.2. Region template data abstraction

The region template abstraction provides a generic container template for common data types, such as pixels, points, arrays (e.g., images or 3D volumes), segmented and annotated objects and regions, that are defined in a spatial and temporal

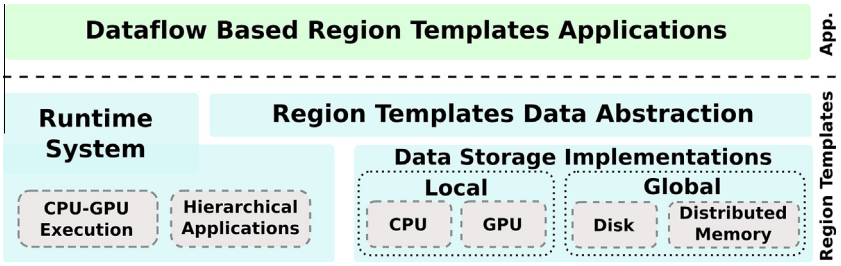


Fig. 2. The architecture of the region template framework. The framework supports the representation and execution of applications using a hierarchical dataflow model with support for hybrid systems equipped with CPUs and GPUs. Data exchanged between application dataflow components are expressed using the region template data abstraction that supports data types commonly used in applications that process data represented in a spatial and temporal domain. Multiple storage implementations of the data abstraction are provided: *Local Data Storage* which manages data in GPU and CPU memories and *Global Data Storage* that manages data across a cluster machine and on disk.

domain. A region template instance represents a container for a region defined by a spatial and temporal bounding box. A *data region* object is a storage materialization of data types and contains the data elements in the region contained by a region template instance. A region template instance may have multiple data regions. Application operations interact with data region objects to retrieve and store data. That is, an application writes data outputs to data regions and reads data inputs from data regions, rather than reading from or writing directly to disk or directly receiving data from or sending data to another stage in the workflow.

Region templates and data regions can be related to other region templates and data regions. Data regions corresponding to the same spatial area may contain different data types and data products. For example, data regions can be related to each other to express structures at different scales that occupy the same space. Data regions can also be related to each other to express evolution of structures and features over time. The spatial and temporal relationship information may be used by the middleware layer to make decisions regarding distribution and management of data regions in a high performance computing environment. Data regions are identified by a *(namespace::key, type, timestamp, version number)* tuple. This identifier intends to provide temporal relationships among data regions related to the same spatial area.

The region template library provides mechanisms for defining region templates and instantiating data regions with support for associative queries and direct access to data elements. A single region template instance may contain multiple data regions. A simplified version of the Region Template class definition is presented in Fig. 3(a). Multiple data regions are stored into a map of data regions. Data regions with the same name are stored into a list. They must differ by at least one of their identifiers: type, timestamp, and version number. A given region template instance also contains a bounding box with the coordinates of the space it covers. As data regions are inserted into a region template instance, the bounding box is updated such that it encapsulates the bounding boxes of all the data regions inserted.

The framework implements two strategies for the instantiation of a data region stored in a region template instance. In the *lazy* strategy, data stored in each data region is only retrieved and the necessary space is allocated when the data is first accessed. Only space for metadata describing the data regions is pre-allocated. The metadata is used to retrieve the actual data from a global data storage implementation. The lazy strategy is used by default in the Master component of an application. In the *immediate* strategy, data in a data region used by an application component is retrieved and the necessary space is allocated before the component is executed. This strategy is used by Workers before scheduling fine-grain tasks to computing devices in a node. The lazy strategy is also employed when, for example, data accessed by a component does not fit entirely in memory or if the bounding box of the data is known at runtime only. In both lazy and immediate strategies, data retrieved during a data region instantiation is within the data region's bounding box. This approach allows for a region template to store metadata for large data regions, which then can be split for parallel computation by modifying the bounding box.

```
class RegionTemplate {
private:
  // Data regions: 1D/2D/3D regular
  // or irregular, and polygons
  std::map<std::string,
     std::list<DataRegion*> >
     templateRegions;

  // Region template name identifier
  std::string name;

  // Region template coordinates
  BoundingBox bb;

  // If false, data are automatically
  // read during component creation
  bool lazyRead;
          ...
public:
  bool insertDataRegion(
    DataRegion *dataRegion);
  DataRegion *getDataRegion(
    std::string namespace,
    int timestamp=0, int version=0,
    int type);

  int getNumDataRegions();
  DataRegion *getDataRegion(int idx);
  void instantiateRegion();
          ...
};
```

(a) Region Template Class.

```
class DataRegion {
private:
  // Type of each data element CHAR, UCHAR,...
  int elementType;
  // Dense, sparse, (2D/3D),...
  int regionType;
  int version, timestamp;
  // Name and instance identifier
  std::string name, id;
  // Resolution of the region
  int resolution;
  // BB surrounding domain and ROI
  BoundingBox bb, ROI;

  // Associates a given bounding box to an id,
  // for data that may be split into pieces.
  std::vector<std::pair<BoundingBox,
      std::string> > bb2Id;

  // Type of storage used to read: distributed
  //shared memory and high performance disk
  int inputDataSourceType, outpuDataSourcetType;
        ...
public:
  DataRegion();
  virtual ~DataRegion();
  virtual bool instantiateRegion();
  virtual bool write();
  virtual bool empty();
        ...
};
```

(b) Data Region Abstract Class.

Fig. 3. Simplified version of the region template and data region Abstractions. A Region Template data structure may store several data regions, which are distinguished by their tuple identifier. The Data Region defines a base class that is inherited by concrete implementations of different data region types. Abstract methods in the Data Region class include those operations that are type specific and need to be implemented to create a new data region type. The system currently includes implementations of the following data region types: 1D, 2D, or 3D dense or sparse regions, polygons.

Access to a data region is done through get/set operations and the data region tuple identifier. The implementations of different data regions types must inherit from the DataRegion abstract class (presented in Fig. 3(b)). In addition to the tuple identifier, the DataRegion class includes attributes such as the type of elements stored, type of the region, etc. The type of the region, for instance, describes whether the data region stores 1D, 2D, or 3D dense or sparse regions, polygons, etc. For each of these types, we have a different implementation of the DataRegion class, because the methods for instantiating/writing the different types from/to the global data storage differ based on the data structure used.

Data region types currently supported in our system are implemented using the OpenCV library [9]. Like ITK [32,33], OpenCV supports a number of data types that include Points, Matrices, Sparse Matrices, etc. An additional feature of OpenCV is the support for GPUs, which includes some of the data structures and processing methods employed in our target applications.

The region template framework provides two types of storage containers. The *local storage* container is used to manage data in CPU and GPU memories on a node. Region template data structures have both CPU and GPU based counterpart implementations, allowing for region template instances and their data structures to reside in the local CPU or CPU memory on a node. The implementations include capabilities for moving data structures between CPU and GPU memories via data upload and download interfaces. Data transfers in each direction may be carried synchronously or asynchronously. The region template interface includes blocking and non-blocking operations to check and verify whether a data transfer has finished. As discussed in Section 3.3.1, the runtime system takes advantage of asynchronous transfer mechanisms to overlap data transfers with useful computation. The *global storage* container manages data across a distributed-memory cluster machine, allowing exchange of data regions, referred to here as *global data regions*, between application components running on different nodes. Our current implementation provides two global storage container versions, one for high performance I/O to disk storage and the other for data management and staging in memory distributed across multiple nodes. These versions are detailed in Section 4. Both global storage versions can co-exist in an application. There are cases in which the use of a high performance disk based mechanism is desirable, for instance, if the application needs to persist data exchanged among components for further analysis. If the goal is to transfer data as quickly as possible among application stages, the distributed memory version tends to be the better choice. The versions can be interchanged by the application user to optimize performance. Multiple global storage versions and the flexible choice of the versions based on application requirements are inspired by the notion of transport methods in the ADIOS framework [42] and the DataSpaces framework [19].

3.3. Runtime system

The runtime system is built on our previous work [63,61]. In this section, we present the core features of the runtime system and the extensions implemented to handle execution of region template applications.

The processing structure of a region template application is expressed as a hierarchical dataflow graph. This representation draws from the filter-stream model implemented in DataCutter [6,57]. Filter-stream applications are decomposed into components, connected to each other through logical streams; a component reads data from one or more streams, carries out data transformations, and writes the results to one or more streams. The representation of region template applications extends this model by allowing an operation itself to be composed of lower-level operations organized into a dataflow. Multiple levels of hierarchy is allowed. The microscopy image analysis application, for example, is expressed as a two level dataflow graph. The first level is the coarse-grain operations level, which represents the main stages of the application. The fine-grain operations level is the second level and represents lower-level operations, from which a main stage is created. Fig. 4 illustrates the two-level hierarchical dataflow representation.

The hierarchical dataflow representation allows for different scheduling strategies to be used at each level. Fine-grain tasks can be dispatched for execution with a scheduler on each computation node, which is more flexible then describing each dataflow component as a single task that should be completely executed using a single device. With this strategy, it is possible to exploit performance variability across fine-grain tasks and to better use available devices.

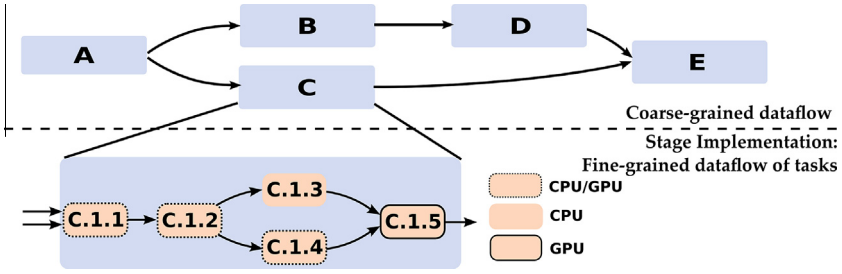


Fig. 4. Hierarchical dataflow model with two levels. Each stage of an analysis pipeline may be expressed as another graph of fine-grain operations. This results in a hierarchical (two-level) computation graph. During execution, stages are mapped to a computation node. Fine-grain operations are dispatched as tasks and scheduled for execution with CPUs and GPUs on that node.

Our runtime system implements a Manager–Worker execution model that combines a bag-of-tasks execution with the dataflow pattern. The application Manager creates instances of coarse-grain stages, which include *input data regions*, and exports the dependencies between the stage instances. The dependency graph is not necessarily known prior to execution; it may be built incrementally at runtime, since a stage may create other stage instances. The assignment of work from the Manager to Worker nodes is performed at the granularity of a stage instance. The Manager schedules stage instances to Workers in a demand-driven basis, and Workers repeatedly request work until all applications stage instances have been executed (see Fig. 5). Each Worker may execute several stage instances concurrently in order to take advantage of multiple computing devices available in a node. Communication between the Manager and Workers is implemented using MPI. Data is read/written by stage components using global data regions, which are implemented from region templates and enable inter-stage communication. Once a stage is received and instantiated by a Worker, the Worker identifies all region templates used by that stage, allocates memory locally on that node to store the associated data regions, and communicates with the appropriate storage implementation to retrieve those regions. Only after data is ready in the node local memory, the stage instance may start executing. The process of reading data overlaps with computation, since tasks created by other stage instances may be concurrently executing with data transfer for the current stage.

Each Worker process is able to use multiple computing devices in a node. The devices are used cooperatively by dispatching fine-grain tasks for execution in a CPU core or a co-processor. Because multiple stage instances may be active on the same node, the tasks may have been created by different stages. In our implementation, fine-grain tasks created by a stage are dispatched for execution by the Worker Resource Manager (WRM) on each node (see Fig. 6 for details). The WRM instantiates one computing thread for each CPU core and each co-processor. Whenever idle, the threads inform the WRM, which selects one of the tasks ready for execution and assigns it to that thread. The scheduling policies used for selecting tasks are described in Section 3.3.1.

When all tasks dispatched for execution by a stage instance have finished, a callback function is invoked to notify a Worker Coordinator Thread (WCT) running on the same node. The WCT writes output region template instances to appropriate global data storage. A message is then sent to the Manager with the information about the completed stage instance. The Manager releases dependencies on that stage instance. As a consequence, other stage instances may be dispatched for execution. The stages of an application are implemented as a specialization of a region template stage abstract class. This class includes interfaces to insert, retrieve, and query region templates used by the application stage. The runtime system also uses these interfaces to access region templates associated with a given stage instance to (i) read/write global data regions that are consumed/produced by the stage instance and (ii) delete region templates that will no longer be used. The region template stage class provides mechanisms for packing/unpacking itself to/from a data buffer when it is assigned to a Worker.

3.3.1. Optimized execution on hybrid systems

This section details the optimizations implemented in our runtime system targeting hybrid machines. The optimizations include smart task scheduling and strategies to reduce impact of data transfers between CPUs and GPUs.

Performance Aware Task Scheduling (PATS). The coarse-grain stages of an application may create several fine-grain task or operations, which are dispatched for execution. An application stage will be composed of several fine-grain tasks which will differ in terms of data access pattern and computation intensity. Thus, the tasks are likely to attain different speedups when executed on a co-processor. In order to take such performance variability into account, we developed the PATS scheduling [63,61]. PATS assigns tasks to a CPU core or a GPU based on the tasks estimated acceleration on each device and on the device load. Once tasks are dispatched for execution with the WRM, they are either inserted in a list of tasks pending or ready-to-execute. The pending tasks are those that do not have all dependencies resolved. New tasks may be inserted in the ready

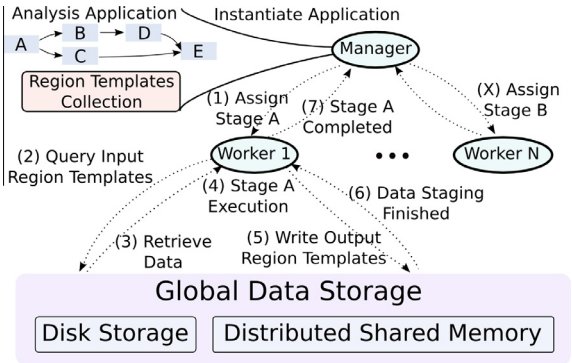


Fig. 5. Overview of the execution model. The execution model is built on top of a Manager–Worker model, which combines the dataflow pattern with a bag-of-tasks style execution. The application developer implements a part of the Manager module that instantiates the application workflow. The Manager creates as many instances of each stage as necessary and sets dependencies between them. During execution, the Manager assigns stage instances for computation with Worker nodes in a demand-driven basis.

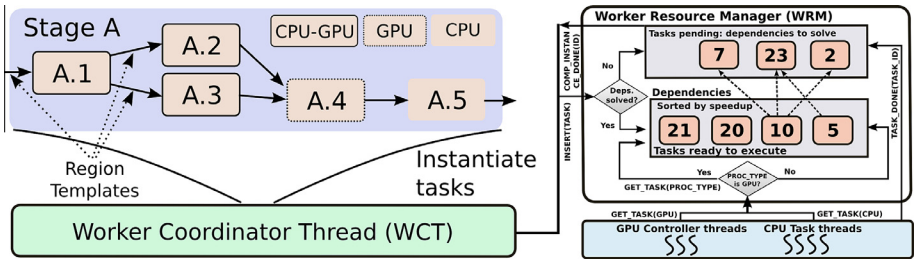


Fig. 6. Each stage of an analysis application may be expressed as another graph of finer grain functions. Functions within a stage instance mapped to a computation node are dispatched as tasks and scheduled for execution by the Worker Resource Manager (WRM) on that node. The WRM creates one computing thread to manage each CPU core or co-processor and assigns tasks for concurrent execution in the available devices.

tasks queue during the execution by the application or because they had the dependencies resolved. The PATS scheduler maintains the list of tasks ready for execution sorted according to the estimated speedup on a GPU. The mapping of tasks to a CPU core or a GPU is performed in a demand-driven basis when these devices become idle. If the idle processor is a CPU core, the task with the smallest speedup is selected for computation, whereas the task with the largest speedup is chosen when a GPU is available (see Fig. 6).

There are several recent efforts on task scheduling for hybrid machines [29,41,43,5,18,8,53,31,59,11,54,58,4,64,27,28,65,30,24,60]. Most of the previous works deal with task mapping for applications in which operations attain similar speedups when executed on a GPU vs a CPU. PATS, on the other hand, exploits performance variability to better use heterogeneous processors. Time-based schedulers (i.e., heterogeneous earliest finish time) have been successfully used in heterogeneous systems for a long time. This class of schedulers, however, is difficult to employ in our target applications. Several operations (ReconToNuclei, AreaThreshold, FillHolles, PreWatershed, Watershed, etc) in our use case application, for example, have irregular data-access and computation patterns with execution times that are data-dependent. The execution times of those operations cannot be accurately estimated before execution. The speedup based scheduling used with PATS has shown to be easier to apply, because we observed smaller variation in speedups incurred as a result of input data variation.

Data Locality Conscious Task Assignment (DL). Time spent transferring data between CPU and GPU memories may have a great impact on the overall execution time of a computation carried out in a GPU. It is important for the scheduler to consider the location of the data used by a task (CPU or GPU memory) in order to maximize performance. It should also minimize data movements without resulting in under-utilization of devices. The region template framework provides a task class API for querying metadata about data regions and region templates associated with a task. Using this API, we have extended the basic scheduler to incorporate data location awareness as an optional optimization. If this optimization is enabled, the scheduler searches the dependency graph of each task executed on a GPU to identify tasks that are ready for execution and can reuse data generated by the current task. The DL optimization in the PATS scheduler compares the performance of tasks that reuse data with those that do not. The task with the largest speedup in the queue (S_q) is compared to the task with the largest speedup that reuses data (S_d). If ($S_d \geqslant S_q \times (1 - \mathit{TransferImpact})$), the task that reuses data is chosen. TransferImpact refers to the portion of data transfer time in the total execution time of a task – this value is currently provided by the application developer or user. This formula aims to take into account both data reuse and higher device utilization. The DL optimization is also employed in CPU-based executions to allow for architecture-aware scheduling of tasks, which is an important optimization in the context of current non-uniform memory architectures (NUMA) machines. During the assignment of a new task for a CPU computation, the tasks depending on the previously computed task are given a priority for scheduling on that CPU core. In our implementation, the dependency graph of the current tasks is explored to find the task that maximizes the amount of data reuse.

Data Prefetching and Asynchronous Data Copy. Data prefetching and Asynchronous Data Copy are other techniques employed by our runtime system to mitigate the costs of data transfers. When DL is not able to avoid data movements, the runtime system will try to perform these operations in background to useful computation. In this approach, input data for tasks to be executed on a GPU or output data by previous tasks are transferred to/from GPU memory while another task is being executed. In our implementation, the dataflow structure of the application is used to identify the data regions that need to be transferred. The region template API is used by the runtime system to perform the actual data transfers. The tasks executed on a GPU are then pipelined through three phases: uploading, processing, and downloading. This three phase execution allows data used by a task to be uploaded while another task is being executed and output data from a third task to be downloaded.

3.4. An example region template application workflow

An application in our system consists of a Manager component and application stages that are executed by Workers. The Manager (Master) component specifies the analysis workflow and defines the region templates for the application. The application developer needs to provide a library of data analysis operations and to implement the portions of the manager that

specify the analysis pipeline and region templates used in each stage. In this process, the Manager component may need to partition each region encapsulated by the region templates among workflow stage instances for parallel execution. Our current implementation supports arbitrary partitions of the data regions.

Fig. 8(a) presents a sketch of the Manager code in the microscopy image analysis application. The Manager code defines the libraries in which the application stages are implemented. Further, it reads a list of image tile file names from an input folder and builds a single region template instance ("Patient") from all tiles. In this example, a data region ("RGB") is created, and each image tile is inserted in the data region with appropriate bounding box of the subdomain it refers to. As presented in Section 3.2, a data region may be partitioned into several chunks of data with associated bounding box (see *bb2Id* structure in Fig. 3(b)).

After the data region is created, the Manager code partitions the data domain of the input data region. The user may create arbitrary partitions that are more appropriate for her application. As an example, two partition approaches could be used with the microscopy image analysis application (see Fig. 7) – the code segment that creates partitions is not shown. In the regular case, input data regions are partitioned into 50×50 tiles. The irregular data partition strategy may be based on computed metrics such as approximate distribution of objects or distribution of tissue regions vs background regions.

The Manager creates a copy of each application stage per the partitioning of the input domain and associates the data regions in each partition with the application stages. The data region information includes the bounding box (region of interest) of that instance, the specification of the data region (e.g., *input*, *output*, or *input and output*), and the global storage container. During this process, dependencies between the application stages are also recorded. Finally, the stage instances are dispatched for scheduling and execution by the runtime system.

Fig. 8(b) presents the code for the segmentation stage in the microscopy image analysis application. A region template, called "Patient", defines a container for data regions "RGB" and "Mask". The data region "RGB" is specified as input and output, meaning it is read from a data storage and written to a storage at the end of the stage. The data region "Mask" is specified as output, since it is produced by the segmentation stage. When a copy of the segmentation stage is executed on a node, the instance interacts with the runtime system through the region template. It requests the data region "RGB", creates the data region "Mask", associates "Mask" with the distributed memory data storage implementation "DMS", and pushes it to the region template instance. We should note that the code in the figure is simplified for space reasons. As described earlier, the runtime system have access to data regions and region templates used by a stage and can instantiate the required data region and make it available to the segmentation stage (see Fig. 5).

In this example, a copy of each stage is executed per partition. Assume the bounding box of the entire region is $(< 0, 0; 99, 99 >)$. The bounding box of partition 4 of the data region "RGB" is defined as $< 50, 50; 99, 99 >$, a copy of the segmentation stage, e.g., "Seg 4" in Fig. 7 is created to compute that region. Preserving the bounding box of the original region template is important to allow for the application to identify the location of the data regions in the original data domain. This information is useful, for instance, for computations involving overlapping borders. The case of ghost cells, for instance, may be handled in a region template application by (i) creating regions of interest (ROIs) that include the ghost cells during the process of reading data and (ii) shrink the ROIs to remove the ghost cells before the data is staged.

As noted earlier, data regions can be associated with different data storage implementations. In the code segment in Fig. 8(b), the data region "RGB" is read from disk ("DISK") and written to the "DMS" storage implementation. If none of the tuple identifiers of the data region are modified, two copies of the same data region could exist in different global storage implementations after segmentation. In this case, unless otherwise specified by the user, the system will use the more

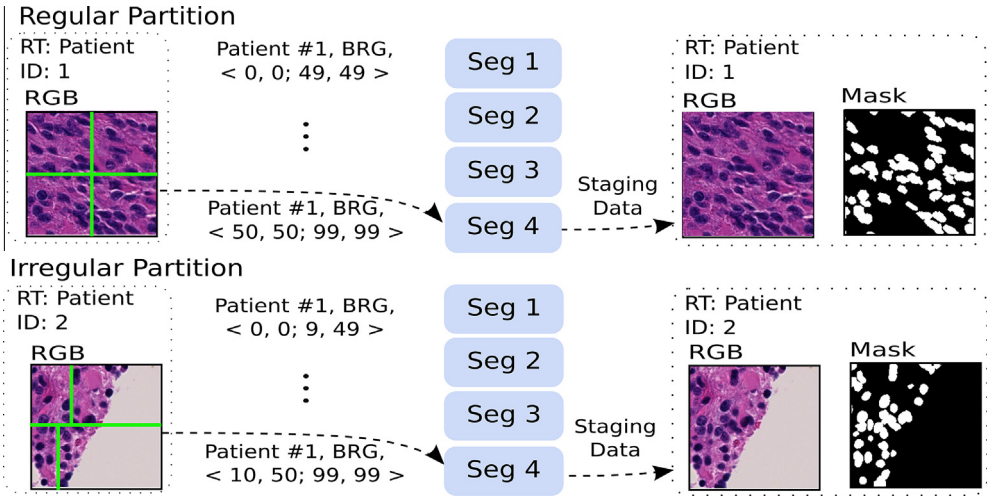


Fig. 7. Two possible data partitions and instantiations of a stage, using regular (blocks of 50×50 pixels) and irregular blocks for better balancing of computational load. Irregular partitions would be useful, e.g., in computations of unstructured grids.

```
int main (int argc, char **argv){
        ...
  RegionTemplate *rt;
  // Libraries that implement components
  SysEnv sysEnv(argc, argv, "libcomponentsrt.so");

  // Create a region templates instance. In this example, a single
  // input image is stored into multiple files. Each file is inserted
  // into the same data region with appropriate bounding box.
  rt = RTFromFiles(inputFolderPath);

  // Create partition of the RT for parallel execution
  vector<BoundingBox *> partitions = createPartition(rt);

  // Instantiate application dependency graph
  for(int i = 0; i < partition.size(); i++){
    // Create an instance of Segmentation stage
    Segmentation *seg = new Segmentation();
    // Insert region template to be used with its ROI:partitions[i]
    seg->addRegionTemplateInstance(rt, "RGB",  partitions[i], INPUT_OUTPUT, DISK);
    seg->addRegionTemplateInstance(rt, "Mask", partitions[i], OUTPUT, DMS);

    // Create instance of Feature Computation stage
    FeatureExtraction *fe =  new FeatureExtraction();
    fe->addRegionTemplateInstance(rt, "RGB",  partitions[i], INPUT, DMS);
    fe->addRegionTemplateInstance(rt, "Mask", partitions[i], INPUT, DMS);

    // Set stage dependencies
    fe->addDependency(seg);
    // Dispatch stages for execution
    sysEnv.executeComponent(seg);
    sysEnv.executeComponent(fe);
  }
  // Start Executing
  sysEnv.startupExecution();
  // Wait until application is done
  sysEnv.finalizeSystem();
        ...
}
```

(a) Manager component of the application.

```
int Segmentation::run()
{
  // Get region template (RT) handler
  RegionTemplate *inRT = getRegionTemplate("Patient");

  // Get data region inside that RT
  DenseDataRegion2D *rgb =
        dynamic_cast<...>(inRt->getDataRegion("RGB"));

  // Change version of the outputted region
  rgb->setVersion(rgb->getVersion()+1);

  // Create output data region
  DenseDataRegion2D *mask =
        new DenseDataRegion2D("Mask", DMS);
  mask->setBb(rgb->getBb());
  inRT->insertDataRegion(mask);

  // Create processing task
  Segmentation(rgb, mask);
}
```

(b) Simplified code for the segmentation stage.

Fig. 8. Simplified code of (a) the manager component and (b) the segmentation stage in our example application. The Manager component is responsible for setting up the runtime system, initializing the metadata of the region templates and data regions, partitioning the data regions for parallel execution, and creating the application graph with appropriate dependencies between the stage instances. The segmentation code illustrates the interactions of an application stage with region templates to retrieve data and transform it. The staging of output data regions is handled by the runtime system.

recently staged data region in the remaining references to the data region. The case in which overlapping areas of a data region are read and staged by multiple stage instances to the same data storage implementation may also lead to synchronization problems. The global data storage implementation always keeps the last staged version of overlapping data regions. The application developer should set dependencies between stages correctly to avoid such issues.

4. Global storage implementations for data regions

We have developed implementations of data regions that represent data structures used in the object segmentation and feature computation operations. Input to an object segmentation operation is an image tile, output from the segmentation operation is a mask array. Input to a feature computation operation is a pair of (image tile, mask array), while the output of a

feature computation operation is a set of feature vectors. Each feature vector is associated with a distinct segmented object. The implementations include local and global storage. The local storage, detailed in Section 3.2, refers to region templates and data regions local to a node stored in CPU and/or GPU memories. The global storage is accessible by any node in the system and stores global regions that are used to exchange data among the analysis stages of an application. The storage implementations for global data regions are presented in Sections 4.1 and 4.2.

4.1. Distributed memory storage

The distributed memory storage (DMS) implementation is built on top of DataSpaces [19]. DataSpaces is an abstraction that implements a semantically specialized virtual shared space for a group of processes. It allows for data stored into this space to be indexed and queried using an interface that provides dynamic and asynchronous interactions. The data to be retrieved/stored is described through key-value pairs that define a bounding box within the application address space. DataSpaces is implemented as a collection of data storage servers running on a set of nodes. An important component of DataSpaces is its distributed hash table (DHT) presented in Fig. 9, which has been designed as an indexing structure for fast lookup and propagation of metadata describing the data stored in the space. In multi-dimensional geometric domains, DataSpaces employs Hilbert space-filling curve (SFC) [7] to map n-dimensional points to an 1-dimensional domain for storage in the DHT.

Our implementation provides a specialized factory object, which can stage data regions from a region template instance to DataSpaces and retrieve data regions to create local instances of the data regions on a node. In the process of staging a region template instance (and the associated data regions) to DataSpaces, a DataSpaces insertion request (formed using the data region tuple identifier and the data region bounding box) is created for each data region in the region template instance. Then, the data into the data regions are packed according to the application domain description and the system dispatches asynchronous insertion requests to the space. Similarly, in the read operation, the read requests are created from an identifier and bounding box describing the portion of the data domain to be retrieved and DataSpaces is queried in background. After data is returned, the system instantiates the local CPU-based data regions and associates them to the region template instance. The application stages are dispatched for execution only after the data is retrieved and the data regions are locally available. The data movement necessary for the execution of a stage instance is performed in background to the executions of other stage instances.

4.2. High performance disk storage

We have developed an implementation for disk storage based on a stream-I/O approach, drawing from filter-stream networks [3,52,38] and data staging [19,1]. The storage implementation assumes that data will be represented and staged to disk in application-defined chunks. In our current implementation of image analysis pipelines, the processing of a set of images in the segmentation and feature computation stages is carried out in image tiles. Output from the segmentation stage is a set of mask arrays, each of which corresponds to an image tile. A data region with *global* scope is used to store these mask arrays on a distributed memory machine. The I/O component provides the data storage abstraction associated with the data region. When a computation node finishes processing data associated with an image tile, it writes the mask array to the data region. The data region passes the output as a data chunk to the data storage abstraction layer. This approach allows us to leverage different I/O configurations and sub-systems. In the current implementation, in addition to POSIX I/O in which each I/O node can write out its buffers independent of other I/O nodes, we have used ADIOS [42] for data output. ADIOS is shown to be efficient, portable, and scalable on supercomputing platforms for a wide range of scientific applications.

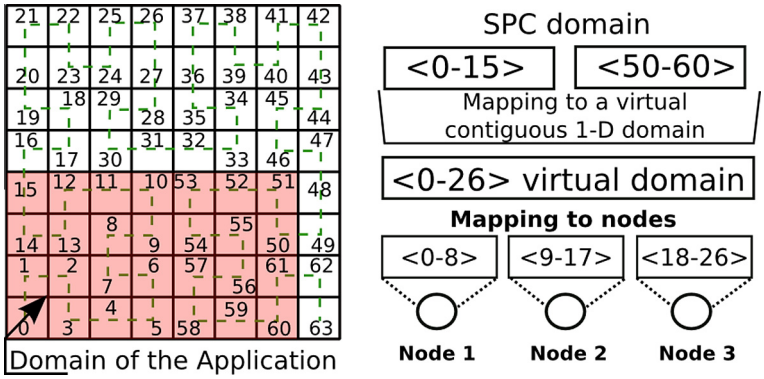


Fig. 9. Space-filling curve (SFC) based distributed hash table (DHT). As presented, the SFC curve maps n-dimensional data elements into 1-dimensional space with locality preserving characteristics. The application-level 2-dimensional domain (highlighted) may consist of a set of subspaces that are non-contiguous in the 1-dimensional transformed space. The non-contiguous subspaces are then mapped into a contiguous virtual domain. This domain is partitioned among storage nodes.

In our implementation, I/O operations can be co-located with computation operations. In addition, a set of CPU cores can be designated as I/O nodes. We have developed support to allow for processors to be partitioned into I/O clusters. All the processors in the same I/O cluster use the same ADIOS group and may need to synchronize. Each I/O cluster can carry out I/O operations independent of other I/O clusters; there is no synchronization across I/O clusters. This clustering is aimed at reducing the synchronization overheads. In the separate I/O-computation configuration, the I/O nodes are coupled to the computation nodes via logical streams. The I/O nodes are further partitioned into groups of k I/O nodes – all the I/O nodes could be in the same group ($k = N$, where N is the number of I/O nodes), or each group could consist of a single I/O node ($k = 1$). In this setting, when a computation node is ready to output a data chunk, it writes the mask array to its output stream. The stream write operation invokes a scheduler which determines to which I/O node the data buffer should be sent, and sends the buffer to the respective I/O node. When an I/O node receives a buffer from its input stream, it puts the buffer into a queue. When the number of buffers in the queue in an I/O node reaches a predefined value, all the I/O nodes in the same group go into a write session and write the buffers out to disk. This implementation facilitates flexibility. The I/O nodes can be placed on different physical processors in the system. For example, if a system had separate machines for I/O purposes, the I/O nodes could be placed on those machines. Moreover, the separation of I/O nodes and computation nodes reduces the impact on computation nodes of synchronizations because of I/O operations and allows a scheduling algorithm to redistribute data across the I/O nodes for I/O balance. We have implemented round-robin and random distribution algorithms. In summary, this module of our system supports separated I/O cores and configurable I/O cluster/group sizes.

5. Experimental results

We have evaluated the region template framework using the Keeneland distributed memory hybrid cluster [67]. Keeneland is a National Science Foundation Track2D Experimental System and has 120 nodes in the current configuration. Each computation node is equipped with a dual socket Intel X5660 2.8 Ghz Westmere processor with hyper-threading disabled, 3 NVIDIA Tesla M2090 (Fermi) GPUs, and 24 GB of DDR3 RAM (Fig. 10). The nodes are connected to each other through a QDR Infiniband switch. The image datasets used in the evaluation were obtained from brain tumor studies [17]. Each image was partitioned into tiles of 4K $\times$ 4K pixels, and the background only tiles were removed from the tile set. The codes were compiled using "gcc 4.4.6", "-O3" optimization flag, OpenCV 2.3.1, and NVIDIA CUDA SDK 4.0. The experiments were repeated 5 times. The standard deviation in performance results was not observed to be higher than 3%. The input tiles were stored in the Lustre file system attached cluster.

5.1. Application implementation

The example application is comprised of a segmentation stage and a feature computation stage, as shown in Fig. 1 (Section 2.2). The segmentation stage receives an RGB image and produces a mask identifying segmented nuclei. The feature computation stage computes a set of shape and texture features for each segmented nucleus. Each stage is composed of a series of functions with the CPU and GPU implementations. For the Morphological Open, Color deconvolution, Canny, and Gradient functions, we used the implementations in OpenCV [9]. The GPU version of Watershed is based on the implementation by Körbes et al. [36]. The ReconToNuclei, FillHoles, and Pre-Watershed functions employ our implementation of the irregular wavefront propagation pattern (IWPP) optimized for GPU execution [62]. The IWPP implementation processes *active elements* in the input domain that contribute to output and uses a hierarchical queue to take advantage of the memory hierarchy of a GPU in order to efficiently track active elements. The connected component labeling function (BWLabel) is implemented by us and employs the union-find pattern described by Oliveira and Lotufo [48]. This pattern first creates a forest in which each pixel is a tree. It then iteratively merges adjacent trees with the same mask pixel value. After the merging phase, object labels are extracted by flattening the trees.

The feature computation stage has relatively regular computation patterns and achieves better GPU acceleration overall than the segmentation stage does. Our implementation of feature computations on objects performs feature calculations within minimum bounding boxes, each of which contains a nucleus. Since each bounding box can be processed independently, a large set of fine-grained tasks are created in this strategy. By restructuring the computations in this way, we avoid unnecessary computation in areas that are not of interest (i.e., that do not have objects) and create a more compact representation of the data.

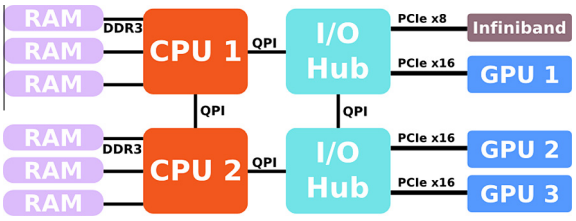


Fig. 10. Keeneland computing node architecture.

The parallel computation of object features using a GPU is done in two steps. In the first step, one GPU block of threads is assigned to each bounding box. The threads in a block collectively compute intermediate results, i.e., the histograms and co-occurrence matrices of the corresponding nucleus. This step takes advantage of the ability of a GPU to dynamically assign thread blocks to GPU multiprocessors in order to reduce load imbalance because of the different sizes, and computational costs, of nuclei. In the second step, feature values per nucleus are calculated from the intermediate results, which are now fixed sized per nucleus. One GPU thread is executed per nucleus in this step. The two-step computation is more efficient than a single-step computation because: (i) the number of threads needed for the computation of intermediate results is much higher than the number of features to compute, resulting in idle threads in a single-step approach; and (ii) the computation of different features by threads in the same block creates divergent branches in the single-step computation.

5.2. Single node performance

These experiments intend to quantify the overhead of using the region template abstraction and the scalability of the example application on a single node. The timings reported in our results are the end-to-end runs and include the I/O costs of reading the input images from the file system.

We have developed a sequential single core version of the application (referred to as *non RT-based*) and compared its execution times with those of the single core version of the application with region templates (referred to as *RT-based*). Both implementations were executed using 3 randomly selected images: Image 1, Image 2, and Image 3, containing 108, 154, and 117 4K × 4K tiles, respectively. As shown in Fig. 11, the non RT-based version is only 1.03× and 1.02× faster, respectively, in the best and average cases. These results show that the overhead of the region template abstraction is very small.

The speedup values of the RT-based version on multi-core CPU and multi-GPU executions, as compared to the single CPU core version, are presented in Table 1. We first evaluated the impact on scalability of the Data Locality Conscious Task Assignment (DL) optimization. With this optimization enabled, when mapping a task to a CPU core, the runtime system will preferably choose a task that has dependency on the task which has just been executed on the same CPU core to increase data reuse and avoid data movement in the memory hierarchy. The multi-core version of the application without the DL optimization achieves sub-linear speedup, i.e., 10.1 on a 12-core configuration. This is a consequence of the application's high memory bandwidth demand. The computation threads compete for the shared memory subsystems and the cost of data retrieval increases as the number of computing cores increases. The use of the DL optimization to minimize the amount of that transfers resulted in a speedup of 10.9× on a 12-core configuration, an improvement of about 1.08× over the version without the DL optimization.

Speedups attained by the multi-GPU executions of the RT-based version are also presented in Table 1. Speedups of 1.94× and 2.82× on two and three GPUs, respectively, are achieved with respect to the single GPU version. This performance was obtained via a careful, architecture-aware placement of threads managing GPUs. In this placement, the GPU manager thread for a GPU is bound to the CPU core that is closest to the GPU in terms of the number of links to be traversed when accessing that GPU. Without this placement, the speedup on 3 GPUs was only 2.27×.

5.3. Disk storage: region template high performance staging to disk

These experiments evaluate the global data storage implementation for high speed staging of data to disk. This module of our system is built using the ADIOS framework as the underlying I/O library. The experimental evaluation was carried out on a large scale cluster, called Jaguar. Jaguar (now known as Titan after upgrades) is a US Department of Energy distributed-memory HPC system. Disk storage is provided through a shared Lustre file system. While the other experiments were carried out using Keeneland, this set of experiments used Jaguar because (1) ADIOS is installed on it for production use, (2) Jaguar is attached to a more scalable storage system, and (3) we were able to use more CPU cores on Jaguar than on Keeneland.

We implemented support for two I/O configurations. In the first configuration, called *co-located I/O*, each CPU core is also an I/O node and performs both computation and I/O operations. In the second configuration, referred to here as *separated I/O*,

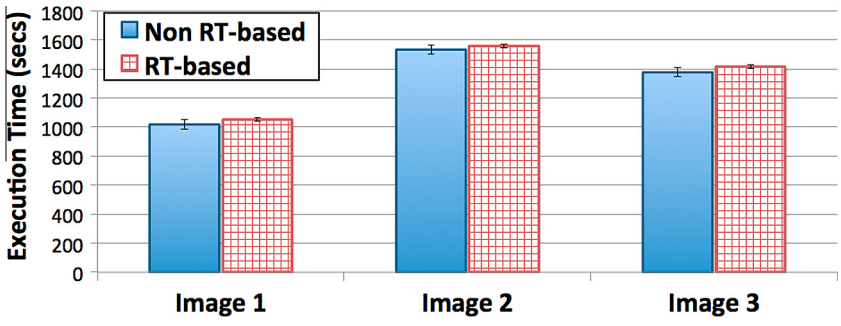


Fig. 11. Comparison of the Non-RT-based and RT-based versions on a single CPU core.

Table 1
Multi-core and multi-GPU scalability of the example application.

# of CPU cores	1	2	4	6	8	10	12
(*a*) *Parallel CPU-based executions with/ without data locality conscious task assignment (DL)*							
Speedup	1	1.9	3.8	5.7	7.5	9.2	10.1
Speedup – DL	1	1.9	3.9	5.8	7.9	9.8	10.9
(*b*) *Multi-GPU scalability*							
# of GPUs	1	2	3				
Speedup	7.9	15.3	22.2				

each CPU core is designated as either a compute core or an I/O core; the compute cores send the output data to the I/O CPU cores for file system writes; these set of I/O CPU cores are distributed across the nodes allocated to the application and may map to the same node as the compute CPU cores. The co-located I/O configuration maximizes the number of cores performing I/O, but introduces synchronization during I/O operations. The separated I/O configuration insulates the compute cores from the synchronization overhead, but introduces additional communication costs. We investigated the effects of using each configuration along with different transport mechanisms supported by ADIOS. Three transport mechanisms were tested: POSIX, where the data are written to the file system independently via standard POSIX calls; MPI LUSTRE, where ADIOS is aware of the Lustre parameters for the file target; and MPI AMR, which is similar to MPI LUSTRE, but a staging layer is introduced to increase data size per I/O request. For each transport mechanism, we partitioned the set of cores participating in I/O into groups of size 1, 15, or the full I/O core size (ALL) to balance between synchronization impact and transport mechanism requirements. The I/O groups are sets of processes that need to synchronize in order to perform I/O operations as the processes in an I/O cluster use the same ADIOS group. The I/O groups were implemented using MPI communicators. An I/O cluster/group can carry out I/O operations independent of other I/O groups; that is, there is no synchronization across I/O groups. For the Separated I/O configuration, we dedicated 60, 512, or 1536 cores for the I/O tasks. Each parameter combination was run in triplicate using 2048 cores with 10240 4K $\times$ 4K input image tiles.

Fig. 12 shows that the co-located I/O configuration performs better than the separated I/O configuration for all experiments. The experiments with the co-located I/O configuration experienced decreased performance when group size was increased for POSIX and MPI LUSTRE, showing that support for smaller I/O groups implemented in the region template framework consistently improved the performance of the application – the default setup supports only the configuration with All processors in one I/O group. For MPI AMR and co-located I/O, we have observed an opposite trend, as smaller groups would perform very poorly due to overheads introduced without staging benefits.

For MPI AMR in the separated I/O configuration, we excluded group size of 1 as this configuration produced extremely poor performance. For most of the separate I/O results, allocating 512 cores to I/O resulted in better performance than 60 or 1536 I/O cores, due to better balancing between data generation rate at the compute cores, data transmission rate between cores, and data consumption rates at the I/O cores. The configurations with 60 cores for I/O resulted in lower

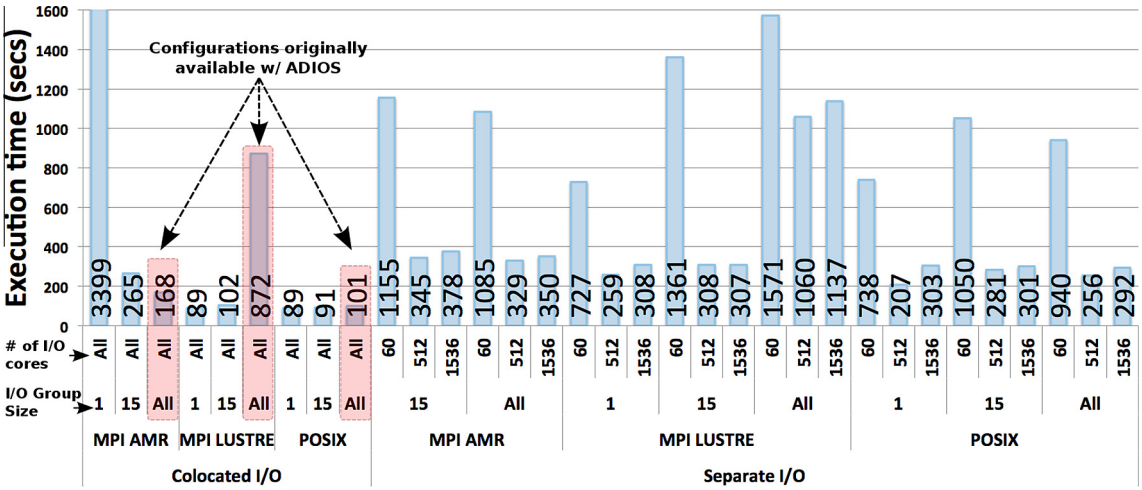


Fig. 12. Evaluation of high performance disk staging implementation on Jaguar. Two configurations are tested for data staging: co-located I/O in which each computing core also performs I/O operations, and separated I/O in which each core is either a compute core or an I/O core. Different transport mechanisms may be used for each configuration: MPI AMR, MPI LUSTRE or POSIX. The I/O group sizes that were evaluated are presented on top of the mechanism chosen. Finally, the number of cores used as I/O cores is presented in the labels closest to the X-axis.

performance, because of communication contention when sending data to the I/O cores. The MPI LUSTRE transport showed a significant decrease in performance with the ALL group size, since it incurs significant synchronization costs.

The use of small I/O group sizes resulted in a speedup of 1.13× as compared to the Co-located I/O, POSIX, and I/O group size ALL setup. We intend to examine in a future work methods for automating the choice of the I/O configuration through the integration with parameter auto-tuning systems [56,68,47].

Even though the separated I/O attained lower performance than co-located I/O, we expect that it can improve the performance of other scenarios. For instance, if the separated I/O processes were executed in storage nodes in the system, it would reduce the communication traffic. In that case, the application would benefit from asynchronous I/O supported by the separated I/O configuration, because it would cache data from I/O operations in memory and perform write operations to storage in background to application execution.

5.4. Performance of distributed memory storage implementation

The experiments in this section evaluate the performance of the distributed memory storage (DMS) and compare it to the high performance disk storage (DISK) to exchange data between the segmentation and feature computation stages. In the DISK storage, I/O nodes and compute nodes were co-located with a group size of 1 (i.e., each compute node performs I/O operations independently of the other nodes) and the POSIX I/O substrate was used for read and write operations. This configuration resulted in the disk storage best performance, as detailed in Section 5.3.

In the experiments the segmentation stage receives a region template with a data region named "RGB" and creates an additional data region named "Mask". The DISK version of the application reads the "RGB" data region from the file system and stages the "Mask" data region to the file system at the end of the segmentation stage. Both data regions are then retrieved from the file system in the feature computation stage. The DMS version also reads the input "RGB" data region from the file system in the segmentation stage. At the end of the segmentation stage, however, the "RGB" data region is staged to DataSpaces along with the "Mask" (the "RGB" data region is marked as INPUT_OUTPUT as is shown in Fig. 8(b)). In the DMS version, hence, the feature computation stage reads both data regions directly from DMS. The results presented in this section are from strong scaling experiments, in which the size of input data and the number of nodes are increased proportionally. A total of 10,800 4K × 4K image tiles are used for the runs on 100 nodes.

The performance of the example application using the DISK and DMS implementations is presented in Fig. 13. As shown, the DMS version achieved better performance in the baseline configuration (4 nodes) and higher scalability when the number of nodes is increased. Fig. 13 shows that the cost of writing the output of the segmentation phase ("Seg. staging") using DMS is at least 10× smaller than that using DISK. We should note that the DMS version writes the "Mask" and "RGB" data regions in this stage, while the DISK version writes the "Mask" only, since the "RGB" data region is already stored in the file system. As a consequence, the amount of data moved by the DMS version is 4× that moved by the DISK version. Although the DISK and DMS versions of the application read the input data regions for the segmentation stage ("Seg. input read") from the file system, the cost of this operation is cheaper in the DMS based executions. This better performance is a side effect of the DMS version not using the file system to exchange data between the segmentation and feature computation stages, which leads to lower load on the file system, and hence to less expensive I/O as compared to the DISK version.

Data transfers rates (GB/s) between the stages of the example application for the DMS implementation are presented in Fig. 14. The results show that very high communication bandwidth is achieved, reaching an aggregate transfer rate of about 200 GB/s. The process of reading data regions for the feature computation stage ("Feature input read") using the DMS version

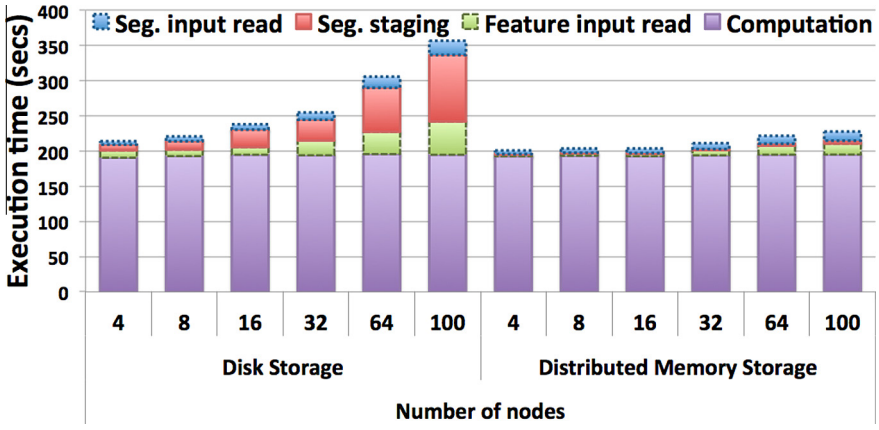


Fig. 13. Efficiency and scalability of high performance disk storage (DISK) and distributed memory (DMS) storage implementations of region templates. In this evaluation, region templates are used to transfer data from the segmentation stage to the feature computation stage of our example application. The cost of the data transfers with the use of disk storage increases quickly as the number of nodes grows, whereas the distributed memory storage based mechanism attains better efficiency and scalability.

is about 3–4× faster than that using the DISK version. This performance improvement is a result of a DataSpaces (used by the DMS) implementation design decision that optimizes data insertion operations. When a process inserts data into DataSpaces, the data is stored on a single DataSpaces server (node) and only the metadata of the data is propagated to the other servers in the system. This scheme avoids data duplication and unnecessary data movement and split. The read operation, on the other hand, may result in a data movement that cannot be avoided and is more expensive.

5.5. Cooperative CPU–GPU executions

In these experiments, we used a fixed set of 104 images (for a total of 6,212 4K × 4K tiles) as the number of nodes is varied. Four versions of the application were executed: (i) *CPUs-only* is the CPU multi-core version that uses all the 12 cores available in each node; (ii) *GPUs-only* uses only the 3 available GPUs in each machine for computation; (iii) *GPUs + CPUs (1L)* uses the CPUs and GPUs in coordination, but the application stages are represented as a single task that bundles all the internal operations; (iii) *GPUs + CPUs (2L)* utilizes CPUs and GPUs in coordination and represents the application as a hierarchical computation graph with two levels. Both cooperative CPU and GPU versions (1L and 2L) can employ FCFS (First-Come, First-Served) or the PATS scheduling strategy. The experiments also evaluate the performance benefits of employing data locality conscious tasks assignment (DL) and data prefetching and asynchronous data copy (Pref.).

Scalability and Scheduler Evaluation. The execution times of the different application implementations are presented in Fig. 15(a). All versions achieved good scalability – the efficiency of the CPUs-only version using 100 nodes was 90%. The GPUs-only version attained a speedup of about 2.25× on top of the CPUs-only, as shown in Fig. 15(b). The cooperative CPU–GPU execution using a single level dataflow graph (GPUs + CPUs (1L)) and FCFS attained a speedup of 2.9× on the CPUs-only version. Fig. 15(a) also presents performance results from the hierarchical version of the application (2L). The 2L configuration with the PATS scheduler was able to significantly (about 1.38×) improve over any other configuration that employs cooperative CPU–GPU execution. PATS used the speedup values presented in Fig. 16, which were collected before the execution in a training phase using a small subset of the data. In addition, "GPUs + CPUs (2L PATS)" achieved a performance improvement of near 4× on top of the multi-core CPUs-only version. The best performance of PATS with 2L is the result of its ability to assign subtasks in a stage to the most appropriate devices, instead of assigning an entire stage for execution on a single processor as in the 1L configuration. Fig. 16 presents the GPU speedups of the individual operations in each stage. We observe that there is a strong variation in the amount of acceleration among the functions, because of their different computation patterns. This variation is taken into account by PATS in scheduling operations to computing devices (CPU cores and GPUs) to improve application performance.

Impact of data movement optimizations. These experiments evaluate the performance benefits of the DL and Pref. optimizations with the 2L PATS version of the application. As is shown in Fig. 15(a), the DL optimization improves the application performance by 5%, because of reduction in the volume of data transferred. When the Pref. optimization is used with the previous version of the application, a speedup of 1.03× is achieved. In the example application, the data transfer times represent 12% of the total execution time. Our optimization techniques were able to eliminate almost 66% of data transfer costs. This reduction along with efficient scheduling of operations across CPUs and GPUs resulted in 8% improvement in overall application execution time. The "GPUs + CPUs (2L PATS + DL + Pref.)" version of the application achieved the best performance with a speedup of 4.34× on top of the multi-core CPUs-only version, as is shown in Fig. 15(b).

Sensitivity to inaccurate speedup estimation. PATS relies on estimated speedup values to keep a sorted queue of tasks. These experiments introduced errors in these values by increasing the estimates for tasks with low speedup values (RBC detection, Morph. Open, AreaThreshold, FillHoles, and BWLabel) and decreasing the estimates for tasks that have higher speedup values. The execution times of the example application using PATS and FCFS schedulers, as the error amount is varied from 10% to 100%, are shown in Fig. 17(a). PATS is impacted by inaccurate speedup estimates, but performance changes are small until the error amounts are high. For instance, if the error in speedup estimate is 50%, the overall performance of the application

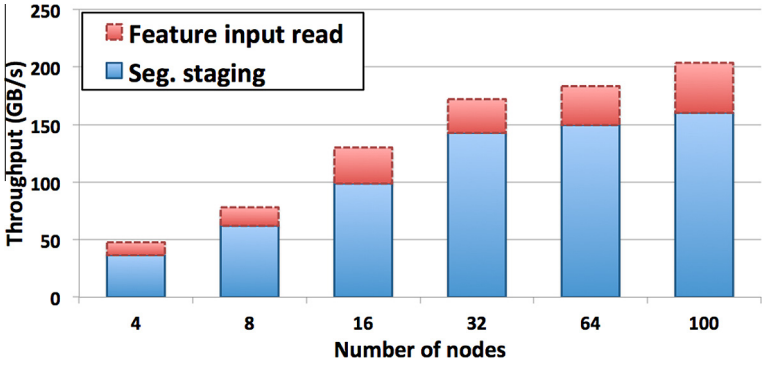
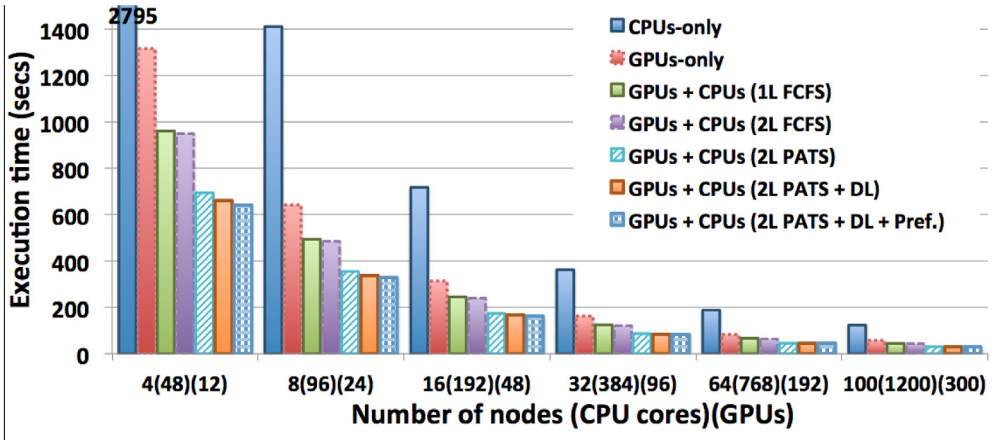
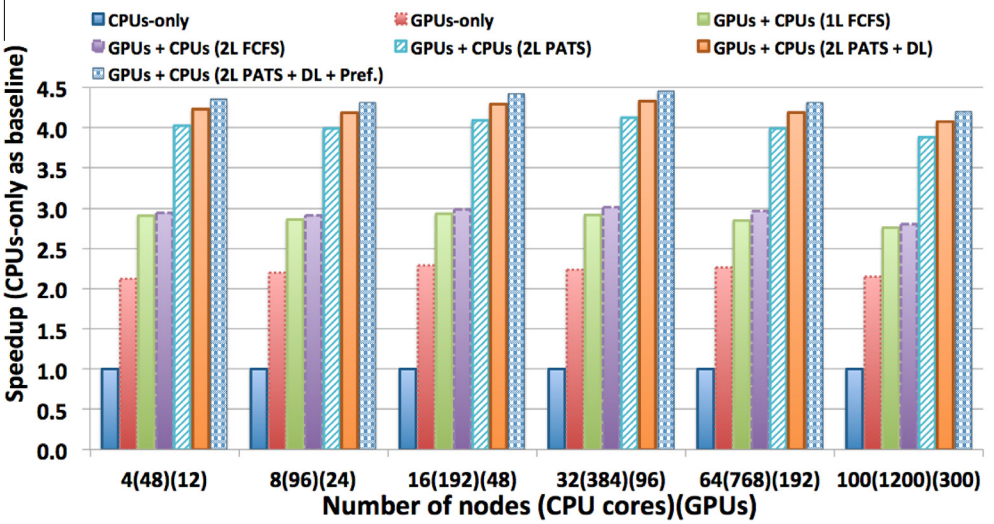


Fig. 14. Data staging/reading throughput of the distributed memory storage (DMS) based global region templates implementation (built on top of DataSpaces) for exchanging data between the segmentation and feature computation stages of the example application.



(a) RT execution time.



(b) Speedup on top of the multicore CPU-only version.

Fig. 15. Cooperative CPU–GPU executions. (i) *CPUs-only* is the CPU multi-core version that uses all the 12 cores available in each node; (ii) *GPUs-only* uses only the 3 available GPUs in each machine; (iii) *GPUs + CPUs (1L)* uses the CPUs and GPUs in coordination and the application stages are represented as using a single task that bundles all the internal operations; (iii) *GPUs + CPUs (2L)* utilizes CPUs and GPUs in coordination and represents the application as a hierarchical computation graph with two levels. The FCFS and PATS scheduling strategies are used in cooperative executions, as well as the locality conscious tasks assignment (DL) and data prefetching and asynchronous data copy (Pref.) optimizations are employed with the best cooperative version of the application.

with PATS decreases only by 8%. PATS is outperformed by FCFS only when there is 100% error in estimates. This level of error corresponds to reversing the order of tasks in the task queue – in that case, the CPU executes the tasks with high speedup values, whereas the GPU receives the tasks with smaller speedup values. In order to better understand the performance of PATS with high estimate errors, we collected the profile of the tasks executed by a GPU when errors of 60%, 70%, and 80% are introduced. Fig. 17(b) presents the percentage of tasks computed by the GPU in each case. The results show a significant increasing in the number of tasks with low speedups executed by GPU as error grows. In addition, other tasks such as "ReconToNuclei" and "Watershed" follow an inverse trend and are assigned in smaller numbers to GPU for computation as the error is increased. Nevertheless, even with 80% estimate errors, PATS is faster than FCFS, since some operations such as "Feature Computation" are correctly dispatched for execution with GPU.

6. Related work

The region template framework leverages data description concepts proposed by Baden et al. [35], as we allow for hierarchical representation of a low dimensional data domain, like representations in adaptive mesh methods. Fortran D [21] and Vienna Fortran [70] propose frameworks for parallel execution of arrays of data elements. Recent projects have developed

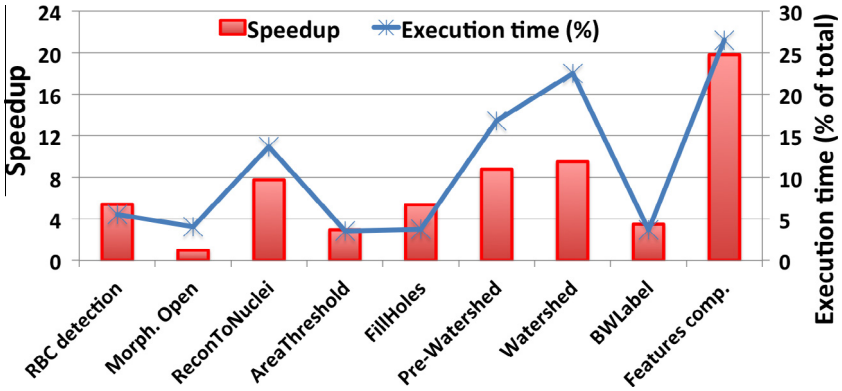


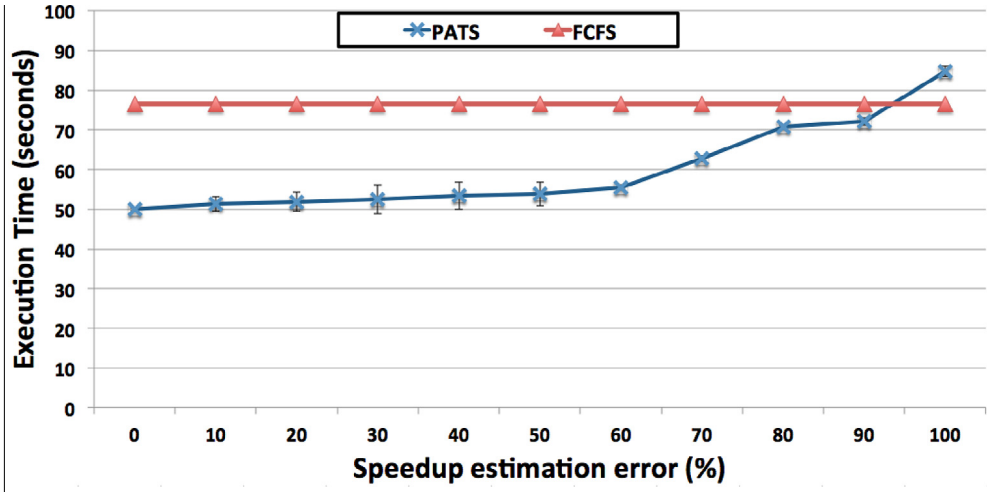
Fig. 16. GPU speedup values for each function in the segmentation and feature computation stages. A strong variation on speedups values is observed due differences in computation and data access patterns. PATS exploits this variation to better use available devices.

data management systems for array based multi-dimensional scientific datasets [10,66,49]. SciDB [10] implements several optimized query operations such as slicing, sub-sampling, and filtering to manipulate large datasets. Pyramid [66] includes similar data types, but is optimized for scalability of metadata management. It also introduces an array-oriented data access model with active storage support, which can be used, for instance, to execute filtering operations or/and data aggregation closer to data sources. Our work differs from these systems in several ways. It enables association of data from multiple sources targeting the same spatial region, which is a common scenario in sensor data analysis where multiple data measurements may be taken for the same region, e.g., measurements of the humidity of certain region over time in monitoring and change detection analysis [13]. It provides a framework for automated management of data across several memory layers on a parallel system, including multiple implementations for efficient I/O, while providing a common interface for data retrieval and storage. Various data types commonly used in microscopy image data are supported. The framework incorporates data management with a runtime system for efficient execution of dataflow applications. Support for data structures with implementations for CPUs and GPUs and scheduling techniques for hybrid systems are other important features provided by the region template framework.

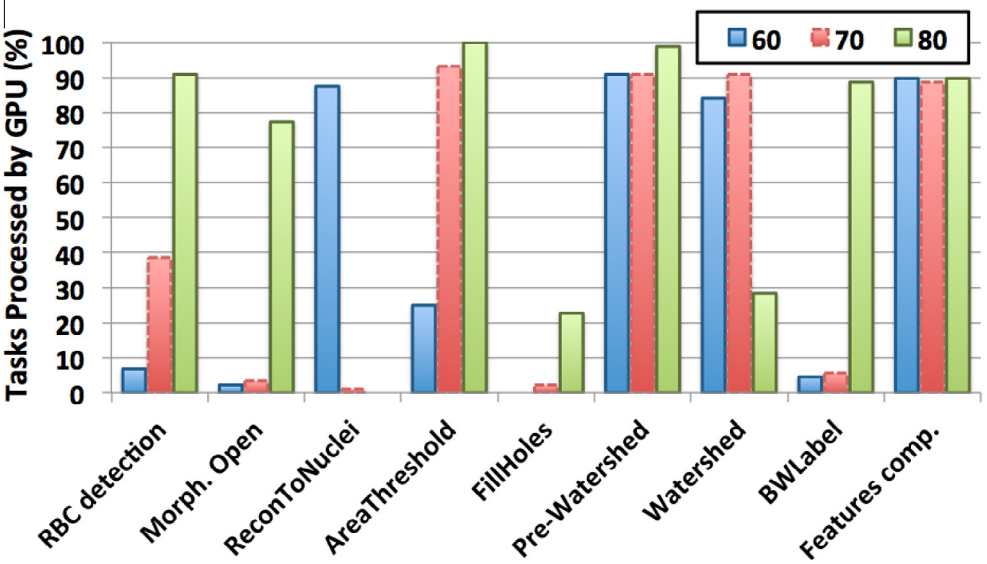
Efficient execution of applications on CPU–GPU equipped platforms has been an objective of several projects [29,41,43,5,18,8,53,31,59,11,54,58,4,64,28,27,65]. The EAVL system [44] is designed to take advantage of CPUs and GPUs for visualization of mesh based simulation output. Ravi et al. [53,31] propose techniques for automatic translation of generalized reductions. The OmpSs [11] supports efficient asynchronous execution of dataflow applications automatically generated by compiler techniques from annotated code. DAGuE [8] and StarPU [5,4] are motivated by regular linear algebra applications on CPU–GPU machines. Applications in these frameworks are represented as a DAG of operations, and different scheduling policies are used for CPU and GPU task assignments. These solutions assume that the computation graphs are static and known prior to execution. This limits their applicability in dynamic applications such as the microscopy image analysis application, because the execution of the next stage may be dependent on a computation result. Our framework allows for the dependency graph to be built at runtime.

KAAPI [23] is a programming framework that supports execution of dataflow applications on distributed memory systems. It is inspired by the Athapascan-1 [22] programming language, which allows for a precise description of dependencies in dataflow graphs, and includes support for asynchronous tasks with explicit shared variables to facilitate data dependency detection. The mapping of tasks onto processors is carried out dynamically using work-stealing techniques. In [30], KAAPI was used and extended to parallelize an iterative physics simulation application on machines equipped with GPUs and CPUs. The authors proposed novel scheduling strategies for dataflows in hybrid systems, which includes runtime load balance with initial workload partition that takes into consideration object affinity. XKaapi [24] has further extended KAAPI. In XKaapi, the programmer develops applications using a multi-versioning scheme in which a processing task may have multiple implementations targeting different computing devices. In order to efficiently execute dataflows on a heterogeneous system, XKaapi implements new scheduling strategies and optimizations for reducing impact of data transfers between devices, locality-aware work stealing, etc. XKaapi is evaluated using regular linear algebra applications. More recently, XKaapi has been extended to support new Intel Xeon Phi coprocessor [40]. XKaapi is contemporary to our runtime system and, as such, it shares a number of optimizations with our system, including strategies for reducing impact of data transfers and locality-aware scheduling. The performance-aware scheduling strategy proposed in our work is not available in other systems, whereas we plan to incorporate the ideas of KAAPI application description to perform implicit calculation of data dependencies in dataflow graphs of region template applications.

Systems such as Linda [12], ThreadMarks [2] and Global Arrays [46] were proposed to provide shared-memory programming model abstractions – typically using matrices to represent data – on distributed platforms with Non-Uniform Memory Access (NUMA). These systems have been successful, because they simplify the deployment of applications on distributed



(a) Impact of inaccurate speedup estimations.



(b) PATS profile: % tasks executed by GPU vs error rate.

Fig. 17. Evaluation of PATS with different amounts of inaccurate speedup estimates. To effectively confound the method and force wrong assignments of operations to computing devices, tasks with low speedups have their speedups increased by a given percentage, whereas tasks with high speedups have their estimates decreased. To understand how high errors in speedup estimates affect scheduling decisions, we also present the profile of tasks executed by the GPU as the error rate is varied from 60% to 80%, which refers to the first interval in which there is a significant performance degradation in PATS.

memory machines and are efficient for applications that exhibit high data locality and access to independent data blocks of a matrix. Data access costs in these systems tend to increase for patterns involving frequent reads and writes of data elements that are not locally stored on a processor. The efficiency and applicability of these solutions is hindered in such cases. Like these systems, our framework provides an abstraction for the representation and computation of large volumes of data on distributed memory machines. However, our framework is not designed as a generic distributed shared memory system in which several processes have access to the entire data domain and consistency is taken care by the system. In our case, data accessed are well defined in terms of input region templates, which allow for data to be moved ahead of computation and restrict data access to locally loaded data.

FlexIO [69] is a middleware that supports flexible data movement between simulation and analytic components in large-scale computing systems. It implements an interface for passing data, originally proposed in ADIOS [42], that mimics "file" manipulations. Data are exchanged during the I/O steps of simulation applications via either disk or memory-to-memory mechanisms. FlexIO supports several I/O options, including (i) "Simulation core": I/O is performed by the application

computing cores; (ii) "Helper Core": cores on the same node receive and stage data from computing cores; (iii) "Staging Core": I/O cores are placed in separate nodes; (iv) "Offline": output simulation data is moved to disk for further analysis. FlexIO also proposes automated heuristics to find the appropriate I/O placement of applications. Placement is static and remains the same during execution. Like FlexIO, the global data storage module of region template supports multiple strategies for placement of computing and I/O cores. However, the region template framework provides data abstractions for spatio-temporal datasets including data types such as sparse and regular arrays, matrices, and objects. These data types are managed by the runtime system potentially using different storage mechanisms. As discussed in Section 4, our implementation of global storage is optimized for asynchronous applications, which unlike simulation applications do not necessarily have synchronization points in which all processes can perform I/O operations. As presented in our experimental results, the use of flexible I/O groups leads to improved performance. In a future work we plan to provide automated placement on distributed system, and the propositions of FlexIO may be adapted for additional optimizations.

7. Conclusions and future work

Researchers have an increasing array of imaging technologies to record detailed pictures of disease morphology at the sub-cellular levels, opening up new potentials for investigating disease mechanisms and improving health care delivery. Large clusters of GPU equipped systems, in which each *hybrid* node contains multiple CPUs and multiple GPUs, have the memory capacity and processing power to enable large imaging studies. However, these systems are generally difficult to program due complexities arising from the heterogeneity of computation devices and multiple levels of memory/storage hierarchies (going from persistent disk-based storage on a parallel file system to distributed memory on the cluster to memories on CPUs and GPUs within a node).

The region template abstraction aims to hide the complexity of managing data across and within memory hierarchies on hybrid systems for microscopy image analysis applications. Some of the characteristics that allow for the efficient execution and data management of region templates are the following: (1) region template applications are instantiated as a graph of computation stages and communication only exists among different stages of the application. Therefore, computation within a given stage uses exclusively local data received as input to the stage; (2) data chunks accessed within a given stage instance are exported to the runtime system, because they are accessed via the region template interface. Therefore, the system knows in advance which data regions (or blocks of a data region) are accessed by a stage and can retrieve the data asynchronously, reducing the impact of data transfer costs; and (3) the mapping of copies of the pipeline stages to computing nodes can be carried by the runtime system in a way to minimize data transfers.

The region template framework provides implementations for common data structures used in target applications; therefore, expect small overhead when developing and integrating new applications.

Our experimental evaluation shows that very high processing and data transfer rates can be achieved in our framework. The processing rate with cooperative CPU–GPU executions using the 2L PATS configuration and 100 nodes is 11,730 4K $\times$ 4K tiles (about 117 whole image slides, 750 GB of data) per minute. The combined data staging and reading rates between the computing stages are about 200 GB/s when distributed memory storage is used. This level of performance will enable much larger imaging studies and is a very promising direction that should lead to better understanding of disease behavior.

We are currently deploying a new biomedical image analysis application on top of region templates. This application computes large-scale cell tracking to study of early stages of metastasis in cancer. The goal of the application is to correlate cell tracking information with other data sources, such as genetic information, in order to better understand the disease. Beyond the great potential science results, this application also brings a new challenging computational scenario. In object tracking, the application only accesses subsets of the data domain that are likely to contain objects of interest. In addition, the path followed by an object until the current timestamp tend to be very useful in identifying in which sub-domain it will be located in future. This context is motivating extensions in region templates to include smart spatial–temporal caching and data prefetching strategies, which could, for instance, anticipate the data reading process and reduce the impact of these operations to the application.

Acknowledgments

This work was supported in part by HHSN261200800001E and 1U24CA180924-01A1 from the NCI, R24HL085343 from the NHLBI, R01LM011119-01 and R01LM009239 from the NLM, RC4MD005964 from the NIH, PHS UL1RR025008 from the NIH CTSA, and CNPq. This work is supported in part by the NIH K25CA181503. This research used resources provided by the XSEDE Science Gateways program and the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the NSF under Contract OCI-0910735.

References

[1] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, F. Zheng, Datastager: scalable data staging services for petascale applications, Cluster Comput. 13 (3) (2010) 277–290.

[2] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwaenepoel, TreadMarks: shared memory computing on networks of workstations, Computer 29 (2) (1996) 18–28.

[3] R.H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D.E. Culler, J.M. Hellerstein, D.A. Patterson, K. Yelick, Cluster I/O with river: making the fast case common, in: IOPADS '99: Input/Output for Parallel and Distributed Systems, Atlanta, GA, 1999.
[4] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, S. Thibault, StarPU-MPI: task programming over clusters of machines enhanced with accelerators, in: S.B. Jesper Larsson Träff, J. Dongarra (Eds.), The 19th European MPI Users' Group Meeting (EuroMPI 2012), LNCS, vol. 7490, Springer, Vienna, Autriche, 2012.
[5] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, Starpu: a unified platform for task scheduling on heterogeneous multicore architectures, in: Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing, 2009, pp. 863–874.
[6] M.D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, J. Saltz, Distributed processing of very large datasets with datacutter, Parallel Comput. 27 (11) (2001) 1457–1478.
[7] T. Bially, A class of dimension changing mapping and its application to bandwidth compression (Phd thesis), Polytechnic Institute of Brooklyn, 1976.
[8] G. Bosilca, A. Bouteiller, T. Herault, P. Lemarinier, N. Saengpatsa, S. Tomov, J. Dongarra, Performance portability of a GPU enabled factorization with the DAGuE framework, in: 2011 IEEE International Conference on Cluster Computing (CLUSTER), 2011, pp. 395–402.
[9] G. Bradski, The OpenCV Library, Dr. Dobb's Journal of Software Tools, 2000.
[10] P.G. Brown, Overview of sciDB: large scale array storage, processing and analysis, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, SIGMOD '10, ACM, New York, NY, USA, 2010, pp. 963–968.
[11] J. Bueno, J. Planas, A. Duran, R. Badia, X. Martorell, E. Ayguade, J. Labarta, Productive programming of GPU clusters with OmpSs, in: 2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS), 2012, pp. 557–568.
[12] N. Carriero, D. Gelernter, Linda in context, Commun. ACM 32 (4) (1989) 444–458.
[13] V. Chandola, R.R. Vatsavai, A scalable gaussian process analysis algorithm for biomass monitoring, Stat. Anal. Data Min. 4 (4) (2011) 430–445.
[14] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, J.H. Saltz, Titan: a high-performance remote sensing database, in: Proceedings of the Thirteenth International Conference on Data Engineering, ICDE '97, IEEE Computer Society, Washington, DC, USA, 1997, pp. 375–384.
[15] L. Cooper, D. Gutman, Q. Long, B. Johnson, S. Cholleti, T. Kurc, J. Saltz, D. Brat, C. Moreno, The proneural molecular signature is enriched in oligodendrogliomas and predicts improved survival among diffuse gliomas, PloS ONE 5 (9) (2010) e12548.
[16] L.A. Cooper, J. Kong, D.A. Gutman, F. Wang, J. Gao, C. Appin, S. Cholleti, T. Pan, A. Sharma, L. Scarpace, T. Mikkelsen, T. Kurc, C.S. Moreno, D.J. Brat, J.H. Saltz, Integrated morphologic analysis for the identification and characterization of disease subtypes, J. Am. Med. Inf. Assoc. 19 (2) (2012) 317–323.
[17] L.A.D. Cooper, J. Kong, D.A. Gutman, F. Wang, S.R. Cholleti, T.C. Pan, P.M. Widener, A. Sharma, T. Mikkelsen, A.E. Flanders, D.L. Rubin, E.G.V. Meir, T.M. Kurc, C.S. Moreno, D.J. Brat, J.H. Saltz, An integrative approach for in silico glioma research, IEEE Trans. Biomed. Eng. 57 (10) (2010) 2617–2621.
[18] G.F. Diamos, S. Yalamanchili, Harmony: an execution model and runtime for heterogeneous many core systems, in: Proceedings of the 17th International Symposium on High Performance Distributed Computing, HPDC '08, ACM, New York, NY, USA, 2008, pp. 197–200.
[19] C. Docan, M. Parashar, S. Klasky, Dataspaces: an interaction and coordination framework for coupled simulation workflows, in: Proceedings of High Performance and Distributed Computing (HPDC), 2010, pp. 25–36.
[20] E.C. Filippi-Chiela, M.M. Oliveira, B. Jurkovski, S.M. Callegari-Jacques, V.D. da Silva, G. Lenz, Nuclear morphometric analysis (NMA): screening of senescence, apoptosis and nuclear irregularities, PloS ONE 7 (8) (2012) e42522.
[21] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, M.-Y. Wu, Fortran D language specification, Dept. of Computer Science, Rice Univ., 1990.
[22] F. Galilee, G. Cavalheiro, J.-L. Roch, M. Doreille, Athapascan-1: On-line building data flow graph in a parallel language, in: Proceedings of International Conference on Parallel Architectures and Compilation Techniques, 1998, pp. 88–95.
[23] T. Gautier, X. Besseron, L. Pigeon, KAAPI: a thread scheduling runtime system for data flow computations on cluster of multi-processors, in: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation, PASCO '07, ACM, New York, NY, USA, 2007, pp. 15–23.
[24] T. Gautier, J.V.F. Lima, N. Maillard, B. Raffin, XKaapi: a runtime system for data-flow task programming on heterogeneous architectures, in: IPDPS '13: Proceedings of the 2013 IEEE International Symposium on Parallel and Distributed Processing, IEEE Computer Society, 2013, pp. 1299–1308.
[25] M.N. Gurcan, T. Pan, H. Shimada, J. Saltz, Image analysis for neuroblastoma classification: segmentation of cell nuclei, in: Conference Proceedings of the IEEE Engineering in Medicine and Biology Society, 2006, pp. 4844–4847.
[26] J. Han, H. Chang, G.V. Fontenay, P.T. Spellman, A. Borowsky, B. Parvin, Molecular bases of morphometric composition in Glioblastoma multiforme, in: 9th IEEE International Symposium on Biomedical Imaging (ISBI '12), 2012, pp. 1631–1634.
[27] T.D. Hartley, U.V. Catalyurek, A. Ruiz, M. Ujaldon, F. Igual, R. Mayo, Biomedical image analysis on a cooperative cluster of gpus and multicores, in: 22nd ACM International Conference on Supercomputing, 2008.
[28] T.D.R. Hartley, E. Saule, Ü.V. Çatalyürek, Automatic dataflow application tuning for heterogeneous systems, in: International Conference on High Performance Computing (HiPC), IEEE, 2010, pp. 1–10.
[29] B. He, W. Fang, Q. Luo, N.K. Govindaraju, T. Wang, Mars: a mapreduce framework on graphics processors, in: Parallel Architectures and Compilation Techniques, 2008.
[30] E. Hermann, B. Raffin, F. Faure, T. Gautier, J. Allard, Multi-GPU and multi-CPU parallelization for interactive physics simulations, in: Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II, Euro-Par'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 235–246.
[31] X. Huo, V. Ravi, G. Agrawal, Porting irregular reductions on heterogeneous CPU–GPU configurations, in: 18th International Conference on High Performance Computing (HiPC), 2011, pp. 1–10.
[32] L. Ibanez, W. Schroeder, L. Ng, J. Cates, The ITK software guide, Kitware Inc, first ed., 2003. ISBN 1-930934-10-6.
[33] H.J. Johnson, M. McCormick, L. Ibánez, T.I.S. Consortium, The ITK Software Guide, Kitware Inc, third ed., 2013.
[34] H. Klie, W. Bangerth, X. Gai, M.F. Wheeler, P.L. Stoffa, M.K. Sen, M. Parashar, Ü.V. Çatalyürek, J.H. Saltz, T.M. Kurç, Models, methods and middleware for grid-enabled multiphysics oil reservoir management, Eng. Comput. (Lond.) 22 (3–4) (2006) 349–370.
[35] S.R. Kohn, S.B. Baden, A parallel software infrastructure for structured adaptive mesh methods, in: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM), Supercomputing '95, ACM, New York, NY, USA, 1995.
[36] A. Körbes, G.B. Vitor, R. de Alencar Lotufo, J.V. Ferreira, Advances on watershed processing on GPU architecture, in: Proceedings of the 10th International Conference on Mathematical Morphology, ISMM'11, 2011.
[37] S. Kothari, A.O. Osunkoya, J.H. Phan, M.D. Wang, Biological interpretation of morphological patterns in histopathological whole-slide images, in: The ACM Conference on Bioinformatics, Computational Biology and Biomedicine, 2012, pp. 218–225.
[38] V.S. Kumar, P. Sadayappan, G. Mehta, K. Vahi, E. Deelman, V. Ratnakar, J. Kim, Y. Gil, M.W. Hall, T.M. Kurc, J.H. Saltz, An integrated framework for performance-based optimization of scientific workflows, in: HPDC, 2009, pp. 177–186.
[39] T.M. Kurç, Ü.V. Çatalyürek, X. Zhang, J.H. Saltz, R. Martino, M.F. Wheeler, M. Peszynska, A. Sussman, C. Hansen, M.K. Sen, R. Seifoullaev, P.L. Stoffa, C. Torres-Verdín, M. Parashar, A simulation and data analysis system for large-scale, data-driven oil reservoir simulation studies, Concurrency – Pract. Exp. 17 (11) (2005) 1441–1467.
[40] J.V.F. Lima, F. Broquedis, T. Gautier, B. Raffin, Preliminary experiments with XKaapi on intel xeon phi coprocessor, in: 25th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD, 2013, pp. 105–112.
[41] M.D. Linderman, J.D. Collins, H. Wang, T.H. Meng, Merge: a programming model for heterogeneous multi-core systems, SIGPLAN Not. 43 (3) (2008) 287–296.
[42] J.F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, C. Jin, Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS), in: CLADE, 2008, pp. 15–24.
[43] C.-K. Luk, S. Hong, H. Kim, Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping, in: 42nd International Symposium on Microarchitecture (MICRO), 2009.

[44] J.S. Meredith, S. Ahern, D. Pugmire, R. Sisneros, EAVL: the extreme-scale analysis and visualization library, in: H. Childs, T. Kuhlen, F. Marton (Eds.), EGPGV, Eurographics Association, 2012, pp. 21–30.
[45] J. Monaco, J. Tomaszewski, M. Feldman, M. Moradi, P. Mousavi, A. Boag, C. Davidson, P. Abolmaesumi, A. Madabhushi, Detection of prostate cancer from whole-mount histology images using Markov random fields, in: Workshop on Microscopic Image Analysis with Applications in Biology (in conjunction with MICCAI), 2008.
[46] J. Nieplocha, R. Harrison, Shared memory programming in metacomputing environments: the global array approach, J. Supercomput. 11 (2) (1997) 119–136.
[47] R. Nishtala, Y. Zheng, P.H. Hargrove, K.A. Yelick, Tuning collective communication for partitioned global address space programming models, Parallel Comput. 37 (9) (2011) 576–591.
[48] V.M.A. Oliveira, R. de Alencar Lotufo, A study on connected components labeling algorithms using GPUs, in: Workshop of Undergraduate Works, XXIII SIBGRAPI, Conference on Graphics, Patterns and Images, 2010.
[49] S. Pallickara, M. Malensek, S. Pallickara, Enabling access to timeseries, geospatial data for on-demand visualization, in: 2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV), 2011, pp. 141–142.
[50] M. Parashar, V. Matossian, W. Bangerth, H. Klie, B. Rutt, T.M. Kurç, Ü.V. Çatalyürek, J.H. Saltz, M.F. Wheeler, Towards dynamic data-driven optimization of oil well placement, Int. Conf. Comput. Sci. 2 (2005) 656–663.
[51] J. Phan, C. Quo, C. Cheng, M. Wang, Multi-scale integration of-omic, imaging, clinical data in biomedical informatics, IEEE Rev. Biomed. Eng. 5 (2012) 74–87.
[52] B. Plale, K. Schwan, Dynamic querying of streaming data with the dQUOB system, IEEE Trans. Parallel Distrib. Syst. 14 (4) (2003) 422–432.
[53] V. Ravi, W. Ma, D. Chiu, G. Agrawal, Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations, in: Proceedings of the 24th ACM International Conference on Supercomputing, 2010, pp. 137–146.
[54] C.J. Rossbach, J. Currey, M. Silberstein, B. Ray, E. Witchel, PTask: operating system abstractions to manage GPUs as compute devices, in: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, 2011, pp. 233–248.
[55] C.T. Shock, C. Chang, B. Moon, A. Acharya, L. Davis, J. Saltz, A. Sussman, The design and evaluation of a high-performance earth science database, Parallel Comput. 24 (1) (1998) 65–89.
[56] C. Tapus, I.-H. Chung, J.K. Hollingsworth, Active harmony: towards automated performance tuning, in: R.C. Giles, D.A. Reed, K. Kelley (Eds.), SC, ACM, 2002, pp. 1–11.
[57] G. Teodoro, D. Fireman, D. Guedes, W.M. Jr., R. Ferreira, Achieving multi-level parallelism in the filter-labeled stream programming model, in: International Conference on Parallel Processing, 2008, pp. 287–294.
[58] G. Teodoro, T. Hartley, U. Catalyurek, R. Ferreira, Optimizing dataflow applications on heterogeneous environments, Cluster Comput. 15 (2012) 125–144.
[59] G. Teodoro, T.D.R. Hartley, U. Catalyurek, R. Ferreira, Run-time optimizations for replicated dataflows on heterogeneous environments, in: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC), 2010, pp. 13–24.
[60] G. Teodoro, T. Kurc, J. Kong, L. Cooper, J. Saltz, Comparative performance analysis of intel (R) Xeon Phi (TM), GPU, CPU: a case study from microscopy image analysis, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS '14, 2014, pp. 1063–1072.
[61] G. Teodoro, T.M. Kurc, T. Pan, L.A. Cooper, J. Kong, P. Widener, J.H. Saltz, Accelerating large scale image analyses on parallel, CPU–GPU equipped systems, in: 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2012, pp. 1093–1104.
[62] G. Teodoro, T. Pan, T. Kurc, J. Kong, L. Cooper, J. Saltz, Efficient irregular wavefront propagation algorithms on hybrid CPU–GPU machines, Parallel Computing, 2013.
[63] G. Teodoro, T. Pan, T.M. Kurc, J. Kong, L.A. Cooper, N. Podhorszki, S. Klasky, J.H. Saltz, High-throughput analysis of large microscopy image datasets on CPU–GPU cluster platforms, in: IPDPS '13: Proceedings of the 2013 IEEE International Symposium on Parallel and Distributed Processing, 2013.
[64] G. Teodoro, R. Sachetto, O. Sertel, M. Gurcan, W.M. Jr., U. Catalyurek, R. Ferreira, Coordinating the use of GPU and CPU for improving performance of compute intensive applications, in: IEEE Cluster, 2009, pp. 1–10.
[65] G. Teodoro, E. Valle, N. Mariano, R. Torres, J. Meira, Wagner, J. Saltz, Approximate similarity search for online multimedia services on distributed CPUGPU platforms, in: VLDB J., 2013, pp. 1–22.
[66] V.-T. Tran, B. Nicolae, G. Antoniu, Towards scalable array-oriented active storage: the pyramid approach, SIGOPS Oper. Syst. Rev. 46 (1) (2012) 19–25.
[67] J.S. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, S. Yalamanchili, Keeneland: bringing heterogeneous GPU computing to the computational science community, Comput. Sci. Eng. 13 (2011).
[68] Q. Yi, K. Seymour, H. You, R.W. Vuduc,D.J. Quinlan, POET: parameterized optimizations for empirical tuning, in:: IPDPS, 2007, pp. 1–8.
[69] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T.-A. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, H. Yu, FlexIO: I/O middleware for location-flexible scientific data analytics, in: 2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS), 2013, pp. 320–331.
[70] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, A. Schwald, Vienna Fortran – a language specification version 1.1. Technical report, 1992.