Published in final edited form as:

Proc IEEE Int Conf Clust Comput. 2017 September ; 2017: 25–35. doi:10.1109/CLUSTER.2017.28.

Parallel and Efficient Sensitivity Analysis of Microscopy Image Segmentation Workflows in Hybrid Systems

Willian Barreiros Jr¹, George Teodoro^{1,2}, Tahsin Kurc^{2,3}, Jun Kong⁴, Alba C. M. A. Melo¹, and Joel Saltz²

¹Department of Computer Science, University of Brasília, Brasília, DF, Brazil

²Department of Biomedical Informatics, Stony Brook University, Stony Brook, NY, USA

³Scientific Data Group, Oak Ridge National Laboratory, Oak Ridge, TN, USA

⁴Biomedical Informatics Department, Emory University, Atlanta, GA, USA

Abstract

We investigate efficient sensitivity analysis (SA) of algorithms that segment and classify image features in a large dataset of high-resolution images. Algorithm SA is the process of evaluating variations of methods and parameter values to quantify differences in the output. A SA can be very compute demanding because it requires re-processing the input dataset several times with different parameters to assess variations in output. In this work, we introduce strategies to efficiently speed up SA via runtime optimizations targeting distributed hybrid systems and reuse of computations from runs with different parameters. We evaluate our approach using a cancer image analysis workflow on a hybrid cluster with 256 nodes, each with an Intel Phi and a dual socket CPU. The SA attained a parallel efficiency of over 90% on 256 nodes. The cooperative execution using the CPUs and the Phi available in each node with smart task assignment strategies resulted in an additional speedup of about 2×. Finally, multi-level computation reuse lead to an additional speedup of up to 2.46× on the parallel version. The level of performance attained with the proposed optimizations will allow the use of SA in large-scale studies.

I. Introduction

We define algorithm sensitivity analysis (SA) as the process of quantifying, comparing, and correlating output from multiple analyses of a dataset computed with variations of an analysis workflow using different input parameters. This process is executed in many phases of scientific research and can be used to quantify the impact of changes in input parameters to differences in the workflow output. The benefits of SA include: (i) better assessment and understanding of correlation between input parameters and analysis output; (ii) ability to reduce the uncertainty/variation of the analysis output by identifying the causes; and (iii) workflow simplification by fixing parameters or removing parts of the code that do not affect the output.

Our work is motivated by image analysis workflows for whole slide tissue images [1]. A typical analysis workflow extracts salient information from tissue images in the form of segmented objects (e.g., cells' nuclei) and their shape and texture features. An example

analysis workflow is presented in Figure 1. Imaging features computed by such workflows contain rich information that can be used to develop morphological models of the specimens to gain insights into disease mechanisms and assess disease progression.

A concern with automated biomedical image analysis is that the output of analysis workflows may be affected by changes in input parameters. Adaptation of SA methods and methodologies employed in other fields [2], [3], [4], [5], [6], can help to better evaluate an image analysis workflow for workflow developers and users. Although the benefits of using SA are many, its use in practice is challenging because of data and computation requirements. For instance, a study using a classic method such as Variance-based Decomposition (VBD) may require hundreds to thousands of runs per parameter of the image analysis workflow. The execution of a single Whole Slide Tissue Image (WSI) will extract 400,000 nuclei on average and can take hours on a single node. A SA study will consider hundreds of WSIs and compute millions of nuclei per run. A single analysis at this scale would take years if executed sequentially.

In order to address the computation challenges of SA, in this work, we leverage the use of large-scale distributed computing systems equipped with accelerators to reduce the execution times of this class of studies. We also propose and evaluate several optimizations targeting efficient data movement, efficient execution on hybrid systems, and simultaneous parameter evaluation in a SA process to eliminate common computations in multiple executions of a workflow as the parameters are varied. The main contributions of this work can be summarized as follows.

- We have designed and implemented a distributed memory platform to execute SA of microscopy image analysis using the Region Templates (RT) framework (described in Section II-A). The execution of our motivating cancer image analysis application [1] on a machine hybrid machine with 256 nodes (a total of 4096 CPU cores and 256 Intel Phi co-processors) demonstrates the scalability of the parallelization, which attained a parallel efficiency of over 90%.
- We have developed optimizations to reuse computations into multiple levels (coarse-grain and fine-grain). The computation reuse occurs because the multiple application runs in a SA may use a common subset of parameters. These optimizations require smart algorithms to identify and maximize the performance gains with reuse. As presented in the results, this optimization accelerates the application in about 2.46 $\times$.
- We have deployed the application in a hybrid machine to evaluate the performance benefits of the cooperative execution using multiple CPU cores and an Intel Phi available in each node of the target system. The performance gains as compared to the execution using only the multi-core CPU is about 2 $\times$ when smart performance-aware scheduling strategies are used.
- We have implemented multiple SA methods (i.e. MOAT and VBD) and have evaluated the performance benefits with different parameter sampling strategies to demonstrate that the performance benefits of our propositions are observed in different studies.

The next section describes the motivating application and the Region Templates (RT) framework used to deploy the application on a parallel machine. Section III describes the SA component in RT. In Section IV, we discuss the proposed algorithms to improve computation reuse and accelerate the sensitivity analysis studies. Section V presents experimental results. We conclude in Section VII.

II. Background

This section describes the motivating application and the Region Template (RT) framework used to deploy the application workflow for execution on distributed systems.

A. Motivating Application

High-resolution microscopy imaging enables the study of disease at the cellular and sub-cellular levels. Investigating the changes in morphology of structures at this level using whole slide tissue specimens can lead to a better understanding of disease mechanisms and a better assessment of response to treatment.

A contemporary digital microscopy scanner can capture from a tissue specimen a whole slide image containing 20 billion pixels (using 40X objective magnification) in a few minutes. An 8-bit color uncompressed representation of this image is over 50GB in size. A scanner with a slide loader can generate hundreds of images in one or two days. Advanced tissue scanning devices are becoming more widely available at low price points for use in research and health care settings. We expect that in the near future research projects will be able to collect tens of thousands of images per study.

A typical analysis workflow consists of normalization, segmentation, feature computation, and final classification operations. The three first analysis stages (Figure 1) are typically the most costly phases of an analysis. The segmentation step, for instance, has to process billions of pixels in a high resolution image and can identify hundreds of thousands of cells. The feature computation step computes 20–50 shape and texture features for each segmented object. All operations on our three most compute intensive stages have been implemented for CPU and Intel Phi. More details on the internal operations of each application stage may be found in our previous work [7], [8].

The final classification phase is less compute expensive and will typically involve the use of data mining algorithms on aggregated information. Some of the targets in this phase include not only classifying tissue images according to progression, but also to gain insights of the underlying biological mechanisms that distinguish between disease subtypes. The effectiveness of an analysis workflow is often sensitive to input data and input parameters. This can lead to concerns about the validity and robustness of extracted or discovered morphological properties. The parameters of the segmentation stage for the motivating workflow along with their description and range values (selected by an application expert) are presented in Table I. The large number of parameters and associated parameters space (21 trillion parameter sets) makes it evident that it would be very hard for one to manually evaluate its sensitivity. This creates a demand for systematic and automated approaches to quantify the impact of the parameters to the output as is provided by SA studies. Since the

segmentation is a critical step in the information extraction, we focus our efforts on the evaluation of this stage of the SA.

B. Region Templates Framework

The region templates (RT) framework supports the execution of dataflow applications [9]. Stages of a workflow in RT consume and produce region template data objects instead of reading/writing data directly from/to other stage or disk. The RT data abstraction used to represent and interchange data consists of storage containers for data structures commonly found in applications that process data in low-dimensional spaces (1D, 2D or 3D spaces) with a temporal component. The data types include: pixels, points, arrays (e.g., images or 3D volumes), segmented and annotated objects and regions, which were implemented using the OpenCV [10] library interfaces to simplify their use.

The main components of the RT framework are: the data abstraction, the runtime system, and the new hierarchical data storage layer (described in detail in Section IV-B). The runtime system supports core functions for scheduling of application stages and transparent data movement and management via the storage layer. The processing structure of RT applications is expressed as a hierarchical dataflow graph. An application stage itself can be composed of lower-level operations organized into another dataflow. The hierarchical dataflow representation allows for different scheduling strategies to be used at each level. Fine-grain scheduling is possible at the second level to exploit variability in performance of application operations in hybrid systems.

The 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, and exports the dependencies between them. The dependency graph 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 using a demand-driven mechanism.

Each Worker uses multiple computing devices in a node by dispatching fine-grain tasks for execution in a CPU core or a co-processor (e.g., Intel Phi). An application stage will be composed of several fine-grain tasks that typically 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 this performance variability into account, we developed the Performance-Aware Task Scheduling (PATS) [9]. PATS assigns tasks to a CPU core or an accelerator based on the tasks estimated acceleration on each device and on the device load. The speedup estimates are provided by the developer and are collected in a profiling phase before the execution.

The PATS scheduler maintains the list of tasks ready for execution sorted according to the estimated speedup on the accelerator. The mapping of tasks to a CPU core or an accelerator is performed in a demand-driven basis when a device becomes idle. If the idle processor is a CPU core, the task with the smallest speedup is selected, whereas the task with the largest speedup is chosen when an accelerator becomes available.

III. Sensitivity Analysis (SA) in RT

An scheme of the SA studies and components that were developed and integrated into the RT framework are illustrated in Figure 2. A SA study in this framework starts with the definition of a given workflow, the parameters to be studied, and the input data. The workflow is then instantiated and executed efficiently in RT using parameters values selected by the SA method. The output of the workflow is compared using a metric selected by the user to measure the difference among a reference segmentation result and the one computed by the workflow using the parameter set generated by the SA method. One of the metrics to measure difference is Dice, which measures the number of pixels identified as objects (foreground) in both the segmented and computed masks. This process continues until the number of workflow runs does not achieve the sample size required by the SA.

A. Sensitivity Analysis Methods

The sensitivity analysis process in our studies can be carried out using a combination of SA methods. Some of the methods implemented in our system include simple screening methods, such as Morris One-At-A-Time (MOAT) design [2] that can be used to quickly identify non-influential parameters, but also other methods to compute more informative importance measures as Pearson's and Spearman's correlation coefficients [11], or the Variance-based Decomposition (VBD) method [3].

These methods are increasingly costly in terms of sampling demands, but also bring more detailed information of correlations between parameters and output. As such, they are employed in different scenarios and a study could use multiple approaches. For instance, a screening method could be employed to prune non-influential parameters before more sampling demanding and expensive methods are used. All these methods need to run an application multiple times as the input parameters are varied. The parameter sets or points selected for evaluation are chosen with probabilistic exploration. The framework supports the commonly used Monte Carlo sampling, Latin hypercube sampling (LHS) [12], quasi-Monte Carlo sampling with Halton or Hammersley sequences, and a few other stochastic methods. These sampling strategies, in special with the low discrepancy sequences, are known to provide a good coverage of the parameters space.

B. Graphical Workflow Description and Deployment in RT

This section describes a new tool for deploying applications/workflows into our framework. The motivation for the development of this component is twofold: (i) we wanted to simplify the application deployment to make it available for application domain experts; and, (ii) the need to have more information about the communication structure of the RT workflows and, as a consequence, to automate the optimizations presented in Section IV.

Our approach to model workflows leverages the Taverna Workbench tool [13]. Our example workflow with Normalization, Segmentation, and Difference stages represented in this tool is presented in Figure 3. This representation includes the input parameters used in each of the stages, and the region templates or data elements used or produced by each stage. Each input parameter is further detailed to include its type and range of values.

Since RT supports a hierarchical workflow representation, each computing stage presented in Figure 3 will be similarly described to represent the workflow of tasks that implements the stage. The information exchanged among the tasks have also the form of region templates, but these data elements are managed in memory. Additionally, the user must provide a description of each stage, containing the tasks with the libraries and the function calls that implement them. Given this information, our system can automatically generate the RT codes for the application workflows.

IV. Optimizations for SA Studies

A. Multi-level Computation Reuse

This section presents the optimizations for computational reuse in SA studies. The SA executes the application work-flow multiple times while varying parameters values to correlate them with application output changes. However, these runs may have common computation paths, which could be reused to speed up the analysis. A common computation is one that uses the same input data and parameter values, thus having the same output. Figure 4 shows two schemes for instantiating an application workflow in a SA study in which multiple parameter sets are tested. The *replica based scheme* does not perform reuse, thus instantiating the entire application workflow for each parameter set. The *compact composition scheme*, on the other hand, merges the instances of an application workflow into a single compact workflow graph to reuse common stages in the separate workflow instances.

The hierarchical workflow representation supported in the RT framework leads to opportunities of reuse at the (i) stage and (ii) task levels. If the reuse is analyzed at the stage level, stage instances will be merged to reuse computation only if all input parameters are the same. However, the potential for merging can be improved by evaluating the workflows of tasks that implement a given stage to identify parts of that workflow that are common. In other words, even if the set of parameter values do not match completely among stages instances, there may be parts (tasks) of the stages that are common. It is worth noting that although merging in both levels are conceptually similar, they demand very distinct algorithmic solutions. Merging at the stage level involves eliminating a stage instance and correcting dependencies and data movement among stages. However, when carried out at the task level, the reuse may be subject to other restrictions and a merging may not be possible in some circumstances. For instance, merging at the task level will increase the number of tasks and memory utilization of a stage instance – a given stage instance is executed in a single node. Therefore, it is necessary to limit the amount of merging according to the system memory and, as a consequence, smart decisions need to be made when choosing the stage merging that will lead to the best improvements. The following sections describe our strategies to merge stages and tasks.

1) Stage Level Merging—The stage level merging needs to identify and remove common stage instances and build a compact representation of the workflow, as presented in Algorithm 1. The algorithm receives the application directed workflow graph (appGraph) and parameter sets to be tested as input (parSets) and outputs the compact graph

(comGraph). It iterates over each parameter set (lines 3–5) to instantiate a replica of the application workflow graph with parameters from *set*. It then calls MERGEGRAPH to merge the replica to the compact representation.

Algorithm 1

Compact Graph Construction

```

1: Input: appGraph; parSets;
2: Output: comGraph;
3: for each set  $\in$  parSets do
4:   appGraphInst = INSTANTIATEAPPGRAPH(set);
5:   MERGEGRAPH(appGraphInst.root, comGraph.root);
6: procedure MERGEGRAPH(appVer, comVer)
7:   for each v  $\in$  appVer.children do
8:     if  $\leftarrow (v' \text{ find}(v, \text{comVer.children}))$  then
9:       MERGEGRAPH(v, v');
10:    else
11:      if  $((v' \leftarrow \text{PendingVer.find}(v)) = \emptyset)$  then
12:        v'  $\leftarrow$  clone(v)
13:        v'.depsSolved  $\leftarrow$  1
14:        comVer.children.add(v')
15:        if v'.deps  $\neq$  1 then
16:          PendingVer.insert(v')
17:          MERGEGRAPH(v, v');
18:        else
19:          comVer.children.add(v')
20:          v'.depsSolved  $\leftarrow$  v'.depsSolved+1
21:          if v'.depsSolved == v'.deps then
22:            PendingVer.remove(v')
23:            MERGEGRAPH(v, v')

```

The MERGEGRAPH procedure walks simultaneously in an application workflow graph instance and in the compact representation. If a path in the application workflow graph instance is not found in the latter, it is added to the compact graph. The MERGEGRAPH procedure receives the current set of vertices in the application workflow (appVer) and in the compact graph (comVer) as a parameter and, for each child vertex of the appVer, finds a corresponding vertex in the children of comVer. Each vertex in the graph has a property called *deps*, which refers to its number of dependencies. The find step considers the name of a stage and the parameters used by the stage. If a vertex is found, the path already exists, and the same procedure is called recursively to merge sub-graphs starting with the matched vertices (lines 8–9). When a corresponding vertex is not found in the compact graph, there are two cases to be considered (lines 10–23). In the first one, the searched node does not exist in comGraph. The node is created and added to the compact graph (lines 11–17). To check if this is the case, the algorithm verifies if the node (*v*) has not been already created and added to comGraph as a result of processing another path of the application workflow

that leads to v . This occurs for nodes with multiple dependencies, e.g., D in Figure 4. If the path (A,B,D) is first merged to the compact graph, when C is processed, it should not create another instance of D. Instead, the existing one should be added to the children list as the algorithm does in the second case (lines 19–23). The PendingVer data structure is used as a look-up table to store such nodes with multiple dependencies during graph merging. This algorithm makes k calls to MERGEGRAPH for each *appGraphInst* to be merged, where k is the number of stages of the workflow. The cost of each call is dominated by the *find* operation in the *comVer.children*. The *children* will have a size of up to n or $|parSets|$ in the worst case. By using a hash table to implement children, the find is $\mathcal{O}(1)$. Thus, the insertion of n instances of the workflow in the compact graph is $\mathcal{O}(kn)$.

2) Task Level Merging—The task level merging is applied after the stage level merging and will seek for partial inter-stage (task) reuse. On a merger of two stages s and t , all tasks of s and t , except the common ones, are inserted into a new stage t' . This operation results in a stage t' with more tasks than s or t individually — note that at least one task in s and t is not common, otherwise these stages would have been merged in the stage merging phase. We describe below three strategies for selecting stages that will be merged to reuse computation at this level. For all strategies, the output is a set of buckets containing stages instances to be merged. The *maxBucketSize* is a parameter that defines the maximum number of stage instances than can be assigned to a bucket. The adequate value of *maxBucketSize* is derived from the underling hardware specifications, i.e. memory per core, and the stage instance memory demands. We can approximate the memory utilization if the RT structures are the only storage containers used by the application.

Naïve Merging: This algorithm traverses linearly the list of stage instances for a given application stage to assign *maxBucketSize* consecutive stage instances to a bucket — the *i*th stage instance is stored in bucket with id $i \% \text{maxBucketSize}$. This solution was designed to quickly select stages to be merged, but its efficiency depends on the order in which stage instances are placed into the input list of stage instances. For instance, if similar stages are close together in the list, more computation is likely to be reused.

Smart Recursive Cut Algorithm (SCA): It represents the stages instances to be merged as vertices of a fully connected undirected graph in which an edge weight is the degree of reuse between vertices. Using this representation, the algorithm performs cuts into the graphs to divide it into unconnected subgraphs that fit in a bucket. The cuts are performed such that the amount of reuse lost with a cut is minimized. In more detail, the partition process starts by dividing the graph into 2 subgraphs using a minimum cut algorithm [14]. Still, after the cut, both subgraphs may have more than *maxBucketSize* vertices. In this case, another cut is applied in the subgraph with the largest number of stages, and this is repeated until a viable subgraph (number of stages $\leq \text{maxBucketSize}$) is found. When this occurs, the viable subgraph is removed from the original graph, and the full process is repeated until the graphs with stage instances yet not assigned to a bucket can fit in one.

The number of cuts necessary to compute a single viable subgraph is $\mathcal{O}(n)$ in the worst case. This occurs when each cut returns a subgraph slightly higher than *maxBucketSize* (i.e.,

$maxBucketSize + 1$) and another subgraph with the remaining nodes. The cut then needs to be recomputed – about $n/(maxBucketSize + 1)$ times – on the largest subgraph until we find a viable subgraph. Also, in the worst case, all viable subgraphs may have a single stage and, as such, up to n buckets could be created. Therefore, the algorithm will perform $\mathcal{O}(n^2)$ cuts in the worst case to create all buckets. In our implementation, the min-cut is computed using a Fibonacci heap [14] to speed up the algorithm, and each cut is $\mathcal{O}(E + V \log V)$. Since the graph used is fully connected, the complexity of a single cut in our case is $\mathcal{O}(n^2)$ and, as consequence, the full SCA is $\mathcal{O}(n^4)$. Although the SCA computes good reuse solutions, its use in practice is limited because of the computational complexity. This motivated the proposal of the strategy described below.

Reuse-Tree Merging Algorithm (RTMA): The RTMA is presented in Algorithm 2. It builds a data structure we call reuse-tree to store stage instances such that instances with common tasks share parts of the tree. After it is built, RTMA selects the stage instances to be merged by traversing and modifying the tree in a bottom-up fashion. An example tree is presented in Figure 5 along with the workflow of tasks (internal operations) and parameter sets used for a stage.

The tree construction (line 4 of Algorithm 2) starts with a root (black) node. Stage tasks are then inserted into the tree recursively, so that the first task of a stage is stored in level 1, second in level 2, etc. During the insertion of a task in a level i , either a new node is created if a node with the same parameter values does not exist in level i or the insertion is executed recursively reusing an existing node. In essence, each node of the tree represents an internal task of the workflow of operations (tasks) of a stage.

The RTMA follows with the assignment of stage instances to buckets. In this process, the RTMA creates a list of nodes (*leafsP List*) containing all parents of leaf nodes (Algorithm 2, line 6). The parent nodes are examined in the PRUNELEAFLEVEL to identify those with at least $maxBucketSize$ children. Sibling nodes of the selected parents are grouped in $maxBucketSize$ nodes and assigned to a new bucket, before they are removed from the reuse-tree. Figure 5 shows an example of the tree before and after the pruning. The gray sibling nodes were assigned to new buckets and removed from the tree. These new buckets are then included into the *bucketList* (Algorithm 2, line 8).

The MOVEREUSETREEUP is further executed in the pruned tree to move one level up the remaining leaf nodes and, consequently, reduce the tree height. This operation first removes every branch of the tree without a leaf node. This is done by removing every node of the parent list that had all their children assigned to a bucket in the PRUNELEAFLEVEL. The ancestors of the non-leaf node removed are also removed in case they have no other child branch. Following the node removal, the rest of MOVEREUSETREEUP method pushes leaf nodes a level up. This is done by removing the leaf node parents from the tree, and making the leafs as children of their former grandparent. The result of this operations is also presented in Figure 5.

The process of assigning nodes to buckets is repeated while the reuse-tree has height higher than 2 (lines 5–9). After this point, the root node is father of all existing nodes, and its children are assigned in groups of $maxBucketSize$ elements to new buckets. One may have noticed that the RTMA assumes that the internal tasks of a stage are organized in a pipeline, while our runtime system supports the execution of directed workflows of tasks. If the tasks are represented as workflows as shown in Figure 4 in which tasks B and C can execute concurrently, the input workflow will be transformed into a pipeline for sake of the RTMA analysis. In this transformation, multiple computation paths (those starting from a single tasks, i.e. B and C) are interpreted as if they were a pipeline in which C comes after B. Please, note that the code generation phase (See Figure 2) that follows the RTMA analysis considers the original workflow of tasks to create the compact graph representation that will be executed. As such, the multiple concurrent computation paths will still be executed in parallel.

Algorithm 2

Reuse-Tree Merging Algorithm (RTMA)

```

1: Input: stages; maxBSize;  $\triangleright$ maxBSize refers to maxBucketSize
2: Output: bucketList;
3: bucketList  $\leftarrow \emptyset$ ;
4: rTree G  $\leftarrow$  GENERATEREUSEETREE(stages)
5: while rTree.height  $> 2$  do
6:   leafsPList  $\leftarrow$  GENERATELEAFSPARENTLIST(rTree)
7:   newBuckets  $\leftarrow$  PRUNELEAFLEVEL(rTree, leafsPList, maxBSize)
8:   bucketList  $\leftarrow$  bucketList  $\cup$  newBuckets
9:   MOVEREUSEETREEUP(reuseTree, leafsPList)
10: while rTree.root.children  $\neq \emptyset$  do
11:   bucket  $\leftarrow \emptyset$ 
12:   while rTree.root.children  $\neq \emptyset$  and bucket.size  $\leq$  maxBSize do
13:     newBucket  $\leftarrow$  removeFirstChilden(rTreeRoot);
14:     bucketList  $\leftarrow$  bucketList  $\cup$  bucket
15: return bucketList

```

The RTMA complexity is dominated by operations in lines 4–9. The GENERATEREUSEETREE performs the insertion of n stage instances with k tasks each. In the worst case, all stage instances differ by one task, and a single insertion is $\mathcal{O}(k+n)$ and, as a consequence, the insertion of n instance is $\mathcal{O}(kn+n^2)$. Following, the loop (lines 5–9) executes $k-1$ iterations. The GENERATELEAFSPARENTLIST locates leaf parent nodes with cost $\mathcal{O}(kn)$. The PRUNELEAFLEVEL iterates through the leaf parent list to create buckets, thus is $\mathcal{O}(n)$. The MOVEREUSEETREEUP has a worst case $\mathcal{O}(n)$ that happens when all leaf nodes are moved up. Thus, the $k-1$ loop iterations is $\mathcal{O}(k^2n)$. Thus, RTMA has a complexity $\mathcal{O}(n^2+k^2n)$, which is dominated by the n^2 component since in practice $n \gg k$.

B. Data Storage Layer

SA studies process the same data elements multiple times as parameters are varied. To take advantage of this, we have developed a new hierarchical storage infrastructure for RT and a strategy that considers data locality during the scheduling of application stage instances.

1) Data Storage Architecture and Implementation—The data storage layer is in charge of storing and retrieving instances of region templates. It can use an arbitrary number of memory layers within a node and across a distributed memory system. The memory/storage hierarchy of the target system is defined in a configuration file that includes the number of storage levels, the position of each storage in the hierarchy, and the level description: type of device (RAM, SSD, etc), capacity, and visibility (local or global). Storage specified as *local* can only be directly accessed within the node (Worker process). Storage specified as *global* is visible to other nodes and is used to exchange data among stages of an application.

The runtime system contacts the data storage layer whenever a region template instance is requested by an application stage. If the data is found in a local storage component, it is directly returned to the application. If the data is found in global storage, it is retrieved and transferred by the storage layer to the requesting node. If the data is in the local storage of the node in which the data was produced (the source node), inter-processor communication is necessary with the source node to move the data to global storage, before the data can be retrieved.

The insertion of data regions or region templates is always performed into the highest (i.e., the fastest) level of the hierarchy with enough capacity to save the data. When a level reaches its maximum storage capacity, a cache replacement strategy is employed to select data regions that should be moved to a lower level in the hierarchy. Each level of a storage hierarchy may use one of the supported data replacement policies: First-In, First-Out (FIFO) and Least Recently Used (LRU). New policies can be incorporated via the application programming interface.

2) Coarse-Grained Scheduling for Locality—In order to reduce data access costs, we propose a data locality-aware scheduling (DLAS) approach that considers the location of data to be accessed when scheduling and mapping application stages to the nodes of the computation system.

The DLAS strategy is implemented at the Manager level of the runtime system. When the notification is received indicating that a given application stage instance (referred to as the original stage instance) has finished, the Manager takes into account the locality of the data produced by that stage instance to determine the node in which stage instances that use the produced data should be executed. In this process, DLAS calculates the amount of data reuse of stage instances that consume the data, and inserts them into a queue of preferred stage instances for execution in the Worker node that executed the original stage instance. A queue of preferred instances is maintained for each Worker in decreasing order of the amount of expected data reuse. This amount can be automatically calculated in our system through the input RT data regions interface that includes a method to return the memory utilization.

When a Worker requests a stage instance for execution, the Manager will try to assign the stage instance that reuses the maximum amount of data — that is, the stage instance on the top of the queue of the requesting Worker. If the queue is empty or none of the stage instances in the queue have dependencies resolved (i.e., they cannot be scheduled for execution), an instance is chosen using the First-Come, First-Served (FCFS) order among those ready for execution.

Data-aware scheduling has been employed in a large number of works that range from grid environments to machines with hybrid processors [15], [16], [17], [18]. In this work, we leverage the ideas from the previous work to apply a similar strategy to the problem of SA, which has a strong data reuse component because of its nature of recomputing the same datasets multiple times. We also allow for the use multiple storage levels and the possibility of transparently exploiting these resources in SA studies.

V. Experimental Evaluation

We evaluated the proposed optimizations using a set of tissue images from brain cancer studies [1]. The images were divided into 4K×4K tiles for concurrent execution. The image analysis workflow consisted of normalization, segmentation and comparison stages. The comparison stage computes the difference between masks generated (Dice). The experimental evaluations were conducted on the TACC Stampede cluster. Each node has dual socket Intel Xeon E5-2680 processors, an Intel Xeon Phi SE10P or MIC (Many Integrated Core) co-processor and 32GB RAM. The nodes are inter-connected via Mellanox FDR Infiniband switches. Stampede uses a Lustre file system accessible from all nodes. The application and middleware codes were compiled using Intel Compiler 13.1 with “-O3” flag. The MIC operations used the *offload mode* – a computing core was reserved to run the offload daemon and at most 240 threads were launched on the coprocessor.

A. Performance Benefits of the Cooperative Execution using CPUs and Intel Phi

In this section, we analyze the performance of the application in a hybrid setting with CPUs and MICs. The workflow is implemented as a 2-level hierarchical workflow with the first level being the coarse-grained stages of normalization, segmentation, and comparison. The second level consists of workflows of operations that implement each of the stages as presented in Section II-A. Five versions of the hybrid setup were evaluated: (1) CPU-only uses all CPU cores; (2) MIC-only uses only the co-processors; (3) CPU-MIC FCFS uses the CPU cores and co-processor with distribution of tasks among processors using FCFS (First-Come, First-Served); (4) CPU-MIC HEFT uses the CPU cores and coprocessor with distribution of tasks among processors using HEFT (Heterogeneous Earliest Finish Time); (5) CPU-MIC PATS uses the CPU cores and co-processors with the PATS scheduler for task scheduling.

The evaluation was performed in a weak scaling experiment in which dataset size and computation nodes are increased proportionately. The experiment dataset contains up to 136,568 4K×4K image tiles (6.5TB of uncompressed data) when the number of nodes is 256. The results in Figure 6 show that all versions of the application scaled well and that cooperative execution with the hybrid configuration led to significant performance gains.

Moreover, the use of our PATS scheduler improved the performance on top of FCFS and HEFT on average by about $1.32\times$ and $1.2\times$, respectively. Performance gains from PATS result from the better ability of PATS in taking into account heterogeneity in performances (speedups) of different tasks when assigning tasks to processors.

B. Impact of Multi-level Computation Reuse and SA methods

This section presents the impact of the computation reuse to the performance for the MOAT and VBD SA methods. We first compute MOAT on all the application parameters, because it demands a smaller per parameter sampling to exclude those parameters that are non-influential to the output from the VBD. Most of the experiments in this section were executed using a small number of machines, because this section intended to detail the gains with the reuse optimizations. However, Section V-B3 presents the results for runs with a large number of nodes in which the gains with the computation reuse optimization remain the same.

1) Impact of Multi-level Computation Reuse for MOAT—Figure 7 presents the execution times of MOAT studies with parameter sample sizes varying from 160 to 640, which were executed using only 6 nodes to demonstrate the impact of the optimizations. The parameters were generated with a quasi-Monte Carlo sampling using a Halton sequence, which is known to provide a good coverage of the parameter space. These experiments use *maxBucketSize* set to 7, and the execution times refer to the makespan and include the cost to perform the computation reuse analysis. For the task level merging approaches, the time spent by the merging algorithm is shown in the upper part of the graph bars. Additionally, five application versions were executed: the “No reuse” that employs the replica based composition, the “Stage level” performs reuse only of stage instances, and the “Task Level” that reuses fine-grain tasks and is executed with the Naïve, SCA, and RTMA algorithms.

The results presented in Figure 7 show that all application versions that reused computation significantly outperformed the baseline No reuse version. The Stage Level reached a speedup of up to $1.85\times$ on top of the No reuse, while the application versions with Task Level reuse have higher gains. The Task Level Naïve is only slightly better than the Stage Level ($1.08\times$ faster in the best case). The Task Level with SCA and RTMA have speedups of up to $1.39\times$ and $1.5\times$, respectively, on top of the Stage Level reuse only.

It is also noticeable from Figure 7 that the performance gains with RTMA increase as the sample size grows as a consequence of the better reuse opportunities. With SCA, however, the opposite behavior is observed. This is a result of the higher costs of executing SCA to compute the stages to be merged, which offsets the gains with the actual execution of the application after the merging. The time taken by Naïve, SCA, and RTMA to compute the reuse are shown on the top of their bars on Figure 7. For a sample of size 640, the time taken by SCA is about 26% of the entire execution. It is also interesting to see that although the RTMA takes a much shorter time to compute the merging choices, and provides better solutions than SCA even if the computation reuse analysis time is not considered. In the best case, RTMA attained a speedup of up to $2.61\times$ on top of the “No reuse” version.

2) Impact of Multi-level Computation Reuse for VBD—The performance of the proposed optimizations for the VBD are presented in Figure 8. The VBD was executed using the remaining 8 parameters (the original parameter set contains 15 parameters) that were not discarded in the MOAT analysis. VBD requirements are of the order of hundreds to thousands runs per parameter. As such, the sample size in this experiment is higher and was varied from 2000 to 10000 runs, whereas the same application versions used with MOAT were evaluated. In order to accelerate this analysis, we have increased the number of nodes to 16.

As presented in Figure 8, the relative performance of the application versions is similar to that observed with MOAT, except for the task level merging using SCA. Given that the sample size used in VBD is much higher, the SCA was not even able to finish computing the reuse and begin executing the workflow in 14000 secs.

3) Impact of Bucket Size and Its Effect on Parallelism—Figure 9 additionally presents the impact of varying the *maxBucketSize* size to the execution times. As expected, the increase in *maxBucketSize* leads to smaller execution times because of the larger number of merging opportunities. However, it is interesting to notice that the variation in execution times as a result of the bucket size changes is up to 12%, which shows that the Task Level reuse can achieve significant gains even with small bucket sizes. Finally, in a large-scale SA using with sample size of 240, 4,276 4K×4K image tiles, and 128 computing nodes, using all optimizations and the No reuse, Stage Level, and Task Level RTMA versions of the workflow attained execution times of, 15,681s, 12,544s and 6,173s, respectively.

We want to highlight that the task level merging reduces the number of stage instances in up to *maxBucketSize* and, as a consequence, the parallelism. This could affect the application scalability if the number of stage instances after the merging was not sufficient to completely use the parallel environment. However, this is not the case in a large-scale SA, because the number of stages instances to be executed in a study is very high. For instance, a SA with a sample size of 240 and 4,276 4K×4K image tile, as employed in the previous experiment, involves the execution of $240 \times 4276 \times 3$ (stages) or about 3×10^6 stage instances. Its reduction by *maxBucketSize* during the merging will still lead to a configuration with sufficient stage instances to use the environment in the scale we are running (up to 256 nodes). We have experimentally validated it, and the scalability before and after merging has no significant difference when up to 256 nodes are used. We recognize, however, that if we continue to reduce the stage per node ratio, by either increasing the number of nodes or reducing the number of stages instances, this behavior may change.

C. Performance with Hierarchical Storage

This section presents the application scalability as the configuration of hierarchical storage is varied on the Stampede cluster. We evaluated the storage with 1 level (1L: file system - FS) and 2 levels (2L: RAM+FS), while the data replacement policy is FIFO or LRU. We also analyzed the performance of our data locality-aware coarse-grained scheduling (DLAS) as compared to the FCFS strategy. A dataset containing 6,113 4K×4K image tiles was used.

The performance results presented in Figure 10 show that all versions of the application attained good scalability. The performance of the configuration with a single storage level is faster than the “2L FIFO - FCFS”. This is because the 2L FIFO-FCFS setup has an overhead for maintaining an extra storage level with very low data access hit rate (about 1.5%) in the first level storage (RAM). The performance of the “2L - FIFO - DLAS” configuration is better than the single level versions for all numbers of nodes (1.11× on average). This is a result of higher data access hit rate (up to 72%) in the first level storage (RAM). Finally, the “2L LRU - DLAS” resulted in the best performance with an average 1.15× speedup on the 1L due to improved hit rate (87%).

VI. RELATED WORK

Strategies for computation reuse have already used in other scenarios [19], [20]. The compiler based techniques developed in [19] identify core regions with computations that could be reused during the execution. In [20] the authors propose an approach for reusing computation through the reuse of data products produced across multiple runs of a workflow. In this work, we propose computation reuse with a multi-level strategy that combines (i) the reuse from multiple application runs (Stage Level) and (ii) the fine-grain computation reuse that is achieved by reusing tasks. The reuse of computation in our work is attained through a stage merging process, instead of reusing data from previous executions. With this approach, it is not necessary to manage the storage across application runs and there is no need for additional space in the storage to hold the data. The reuse at the level of tasks proposed in this work requires more sophisticated merging approaches, which have to consider that the number of stage instances that could be merged is limited by the hardware characteristics.

The development of data abstractions for data management on distributed systems is another related topic [21]. The work of Lofstead et al. [21] deals with efficient staging of data to storage systems for scientific applications. In RT, we use of multiple storage layers to process out of core data elements that are partitioned by the application for parallel processing. Our system combines scheduling decisions with data locality information to minimize data movements.

Other runtime systems have been proposed for hybrid machines [22], [23]. The scheduling strategies for hybrid machines mostly focused on mapping applications in which internal operations attain similar speedups when executed on a MIC vs a CPU. Our PATS mapping strategy uses variability in tasks performance to better utilize hybrid machines. In our earlier work [9], [24], we evaluated PATS using a two-stage application analysis pipeline consisting of segmentation and feature computation stages. In this work, the application consists of normalization, segmentation, and comparison stages. We target the execution algorithm SA, and compare the scheduling to time-based approaches.

VII. Conclusions

The execution of SA using large-scale datasets could strongly benefit automated microscopy image analysis, but it has limited use in practice because of the high computational demands.

In this paper, we have leveraged high performance computing platforms equipped with CPUs and accelerators to efficiently execute SA. Our solution includes an efficient parallelization with several optimizations, which consist of the cooperative use of CPUs and Intel Phi, data-aware scheduling, and multi-level computation reuse. The performance evaluation using a complex cancer image analysis application with large datasets on a large-scale cluster system demonstrated that these optimizations resulted in significant performance improvements. We argue that the use of the proposed runtime optimizations can enable systematic, comparative study of analysis pipelines and improve analysis results when large datasets need to be analyzed. As future work, we will develop other algorithms for task level merging, and evaluate the proposed strategies in other application domains. We expect that the optimizations can benefit other applications executing the same SA methods.

Acknowledgments

This work was supported in part by 1U24CA180924-01A1 from the NCI, R01LM011119-01 and R01LM009239 from the NLM, CNPq, Capes/Brazil grant PROCAD-183794, and NIH K25CA181503. This research used resources of the XSEDE Science Gateways program under grant TG-ASC130023.

References

1. Kong J, Cooper LAD, Wang F, Gao J, Teodoro G, Mikkelsen T, Schniederjan MJ, Moreno CS, Saltz JH, Brat DJ. Machine-based morphologic analysis of glioblastoma using whole-slide pathology images uncovers clinically relevant molecular correlates. *PLoS ONE*. 2013
2. Morris MD. Factorial sampling plans for preliminary computational experiments. *Technometrics*. 1991; 33(2):161–174.
3. Weirs VG, Kamm JR, Swiler LP, Tarantola S, Ratto M, Adams BM, Rider WJ, Eldred MS. Sensitivity analysis techniques applied to a system of hyperbolic conservation laws. *Reliability Engineering & System Safety*. 2012; 107:157–170.
4. Campolongo F, Cariboni J, Saltelli A. An effective screening design for sensitivity analysis of large models. *Environmental Modelling & Software*. 2007; 22(10):1509–1518. modelling, computer-assisted simulations, and mapping of dangerous phenomena for hazard assessment.
5. Iooss B, Lemaitre P. A review on global sensitivity analysis methods. *Uncertainty Management in Simulation-Optimization of Complex Systems*, ser Operations Research/-Computer Science Interfaces Series. 2015; 59:101–122.
6. Teodoro G, Kurc T, Taveira LFR, Melo ACMA, Gao Y, Kong J, Saltz J. Algorithm sensitivity analysis and parameter tuning for tissue image segmentation pipelines. *Bioinformatics*. 2017
7. Teodoro G, Kurc T, Kong J, Cooper L, Saltz J. Comparative Performance Analysis of Intel (R) Xeon Phi (TM), GPU, and CPU: A Case Study from Microscopy Image Analysis. *28th IEEE Int Parallel and Distributed Processing Symposium (IPDPS)*. 2014:1063–1072.
8. Kurc T, Qi X, Wang D, Wang F, Teodoro G, Cooper L, Nalisnik M, Yang L, Saltz J, Foran DJ. Scalable analysis of Big pathology image data cohorts using efficient methods and high-performance computing strategies. *BMC bioinformatics*. 2015; 16(1):399. [PubMed: 26627175]
9. Teodoro G, Pan T, Kurc T, Kong J, Cooper L, Klasky S, Saltz J. Region templates: Data representation and management for high-throughput image analysis. *Parallel Computing*. 2014; 40(10):589–610. [PubMed: 26139953]
10. Bradski G. The OpenCV Library. *Dr Dobb's Journal of Software Tools*. 2000
11. Saltelli A., Tarantola S., Campolongo F., Ratto M. *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models*. Wiley; 2004.
12. McKay WJCMD, Beckman RJ. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics*. 1979; 21(2):239–245.
13. Wolstencroft K, et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Research*. 2013

14. Stoer M, Wagner F. A Simple Min-cut Algorithm. J ACM. Jul; 1997 44(4):585–591.
15. Jaykishan, B., Reddy, K Hemant Kumar, Roy, DS. A Data-Aware Scheduling Framework for Parallel Applications in a Cloud Environment. New Delhi: Springer India; 2014. p. 459-463.
16. Becchi, M., Byna, S., Cadambi, S., Chakradhar, S. Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures. New York, NY, USA: ACM; 2010. Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory; p. 82-91. ser SPAA '10
17. Wang K, Qiao K, Sadooghi I, Zhou X, Li T, Lang M, Raicu I. Load-balanced and locality-aware scheduling for data-intensive workloads at extreme scales. Concurrency and Computation: Practice and Experience. 2016; 28(1):70–94. cPE-14-0369.R2.
18. Kosar T, Balman M. A new paradigm: Data-aware scheduling in grid computing. Future Generation Computer Systems. 2009; 25(4):406–413.
19. Ernst MD, Cockrell J, Griswold WG, Notkin D. Dynamically Discovering Likely Program Invariants to Support Program Evolution. Proc of the 21st International Conference on Software Engineering (ICSE) ACM. 1999
20. Wang Y, Li H, Hu M. Reusing Garbage Data for Efficient Workflow Computation. The Computer Journal. 2015; 58(1):110–125.
21. Lofstead JF, Klasky S, Schwan K, Podhorszki N, Jin C. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). CLADE. 2008
22. Bosilca G, Bouteiller A, Herault T, Lemarinier P, Saengpatana N, Tomov S, Dongarra J. Performance Portability of a GPU Enabled Factorization with the DAGuE Framework. IEEE Int Conf on Cluster Computing (CLUSTER). 2011
23. Bueno J, Planas J, Duran A, Badia R, Martorell X, Ayguade E, Labarta J. Productive Programming of GPU Clusters with OmpSs. IEEE 26th Int Parallel Distributed Processing Symposium (IPDPS). May.2012
24. Teodoro G, Kurc T, Andrade G, Kong J, Ferreira R, Saltz J. Application performance analysis and efficient execution on systems with multi-core CPUs, GPUs and MICs: a case study with microscopy image analysis. The International Journal of High Performance Computing Applications. 2017; 31(1):32–51. [PubMed: 28239253]

Figure 1.

Example analysis workflow: normalization, segmentation and feature computation stages are presented with their cascade of internal operations. The parameters used in each of the operations are also shown.

Figure 2.

The parameter study framework. A SA method selects parameters of the analysis workflow, which is executed on a parallel machine. The workflow results are compared to a set of reference results to compute differences in the output. This process is repeated until the sample size (workflow runs) for the SA method is not reached.

Figure 3.
The example workflow described with Workbench.

Figure 4.

Example of a workflow composition. This composition can be performed by either fully replicating the base workflow for each parameter set, or performing a compact composition.

Figure 5. Merging example of 12 stages using the Reuse-Tree algorithm, which is performed by selecting the mergeable nodes, pruning them from the Reuse-Tree and adding them to the solution list, and then moving the remaining nodes up.

Figure 6.
Task scheduling in a weak scaling evaluation.

Figure 7.
Impact of the computation reuse strategies for the MOAT SA method.

Figure 8.
Impact of the computation reuse strategies for the VBD SA method.

Figure 9.
Impact of varying *maxBucketSize*.

Figure 10.
Scalability and performance with different storage and coarse-grained stage scheduling.

Table I

Definition of parameters and range values: parameter space contains about 21 trillion points.

Parameter	Description	Range Values
B/G/R	Background detection thresholds	[210, 220, ..., 240]
T1/T2	Red blood cell thresholds	[2.5, 3.0, ..., 7.5]
G1/G2	Thresholds to identify candidate nuclei	[5, 10, ..., 80] [2, 4, ..., 40]
MinSize(minS)	Candidate nuclei area threshold	[2, 4, ..., 40]
MaxSize(maxS)	Candidate nuclei area threshold	[900, ..., 1500]
MinSizePl (minSPL)	Area threshold before watershed	[5, 10, ..., 80]
MinSizeSeg (maxSS)	Area threshold in final output	[2, 4, ..., 40]
MaxSizeSeg (minSS)	Area threshold in final output	[900, ..., 1500]
FillHoles(FH)	propagation neighborhood	[4-conn, 8-conn]
MorphRecon(RC)	propagation neighborhood	[4-conn, 8-conn]
Watershed(WConn)	propagation neighborhood	[4-conn, 8-conn]