HHS Public Access

Author manuscript

Concurr Comput. Author manuscript; available in PMC 2020 July 25.

Published in final edited form as:

Concurr Comput. 2020 January 25; 32(2): . doi:10.1002/cpe.5403.

Optimizing parameter sensitivity analysis of large-scale microscopy image analysis workflows with multilevel computation reuse

Willian Barreiros Jr[1], **Jeremias Moreira**[1], **Tahsin Kurc**[2,3], **Jun Kong**[4,5,6], **Alba C.M.A. Melo**[1], **Joel H. Saltz**[2], **George Teodoro**[1,2,7]

[1]Department of Computer Science, University of Brasília, Brasília, Brazil

[2]Department of Biomedical Informatics, Stony Brook University, Stony Brook, New York

[3]Scientific Data Group, Oak Ridge National Laboratory, Oak Ridge, Tennessee

[4]Department of Biomedical Informatics, Emory University, Atlanta, Georgia

[5]Department of Computer Science, Emory University, Atlanta, Georgia

[6]Department of Mathematics and Statistics, Georgia State University, Atlanta, Georgia

[7]Department of Computer Science, Federal University of Minas Gerais, Belo Horizonte, Brazil

Summary

Parameter sensitivity analysis (SA) is an effective tool to gain knowledge about complex analysis applications and assess the variability in their analysis results. However, it is an expensive process as it requires the execution of the target application multiple times with a large number of different input parameter values. In this work, we propose optimizations to reduce the overall computation cost of SA in the context of analysis applications that segment high-resolution slide tissue images, ie, images with resolutions of 100k × 100k pixels. Two cost-cutting techniques are combined to efficiently execute SA: use of distributed hybrid systems for parallel execution and computation reuse at multiple levels of an analysis pipeline to reduce the amount of computation. These techniques were evaluated using a cancer image analysis workflow on a hybrid cluster with 256 nodes, each with an Intel Phi and a dual socket CPU. Our parallel execution method attained an efficiency of over 90% on 256 nodes. The hybrid execution on the CPU and Intel Phi improved the performance by 2×. Multilevel computation reuse led to performance gains of over 2.9×.

Keywords

computation reuse; microscopy imaging; parameter sensitivity analysis

Correspondence: George Teodoro, Department of Computer Science, Federal University of Minas Gerais, 31270-901 Belo Horizonte-MG, Brazil. george@dcc.ufmg.br.

1 INTRODUCTION

Parameter sensitivity analysis (SA) examines and quantifies the impact on analysis results of varying the input parameters of an analysis algorithm or application. Moreover, SA is executed by comparing, correlating, and quantifying outputs from multiple application runs as input parameters values are systematically varied. In addition, SA facilitates a better understanding of correlations between input parameters and application output. It can help remove sources of uncertainty, for instance, by re-factoring the application code to eliminate or replace operations associated with parameters of high influence to output variations. It can also simplify parameter tuning by enabling the parameter tuning process to focus on parameters that have the highest impact on output. Finally, SA has been successfully employed in several application domains.[1–6] In this work, however, we target optimizations for efficient execution of SA in image segmentation workflows in whole slide tissue image analysis.

Whole slide tissue images (WSI) capture the sub-cellular structure of tissue in great detail. A typical WSI segmentation workflow locates cells and nuclei in an image, extracts their boundaries, and computes a set of shape and texture features for the segmented nuclei and cells. The segmented objects and features are used in further analyses to look for spatial patterns, correlate with genomic and clinical data, and to study the mechanisms of disease onset and progression.[7–13]

WSI analysis is a complex and compute-expensive process. Figure 1 presents the image analysis workflow employed by the example application used in this work.[7] This workflow is divided into three coarse-grain computation steps: Normalization, Segmentation, and Feature Extraction, which are decomposed into several low-level or fine-grain operations. The computation of each WSI may lead to the identification of about 400 000 nuclei and will execute for hours when carried out on a single machine. Regardless of the number of opportunities and advantages in using information automatically computed from WSIs, there are still aspects that need to be better addressed in order to make these analyses more robust and reliable. For instance, image analysis algorithms are typically parameterized, and changes on input parameters may significantly affect the output results/quality.[14] In this context, understanding the correlations between input parameters and the application outputs is of main importance.

The use of SA in the context of pathology image analysis can lead to a better understanding of the algorithms and improve results in terms of robustness, but its execution is very challenging due to its high computational costs. SA studies typically require the application to be executed several times as parameter values are varied. A method such as Variance Based Decomposition (VBD),[4] which is widely used for SA, may require hundreds to thousands of runs per parameter in order to adequately quantify the correlations between parameters and output. Each run would process hundreds of WSIs in a moderate scale experiment, segmenting hundreds of millions cell nuclei, which, in turn, should be compared to a reference dataset to quantify changes in the output results or objects detected. An experiment at this scale would take years if executed sequentially.[14]

Our work addresses the challenges of executing large scale SA in pathology image analysis workflows. We propose methods for efficient execution on hybrid distributed memory compute systems and novel runtime optimizations targeting SA. In particular, we propose smart computation reuse strategies. The computation reuse opportunities are raised as the application is executed multiple times over the same datasets with different parameter sets. In this setting, there are parts of the application (stages or fine-grain tasks/operations) that may be computed using the same data and parameters values, which leads to the re-execution of these operations or tasks multiple times. In the context of fine-grain operations/tasks within an application coarse-grain stage, computation reuse may be complex since the reuse is attained by merging stage instances to avoid duplicate computations. However, this merging creates coarser-grain stages that may demand more resources (eg, memory). As a consequence, this limits the number of stage instances that could be merged together since a stage instance is executed within the scope of a node. Moreover, the merging impacts the available parallelism due to the smaller number of stage instances available for execution, which, depending on the system scale and number of workflow instances dispatched for execution (ie, the type of the SA method used), may affect the application scalability.

This paper extends our previous work on the domain[15] with the introduction of a computation reuse strategy called Task-Balanced Reuse-Tree Merging Algorithm (TRTMA). TRTMA is able to handle the trade-offs between merging and parallelism availability. As presented in the results, TRTMA has similar or superior performance (up to about 1.7×) than our previous approaches. We also performed additional evaluations to identify the impact of multiple sampling strategies to the reuse, extended the background discussion of sensitivity analysis on the proposed framework, and presented a comparative review of related works. The main contributions of this work can be summarized as follows.

- We present a system for efficient execution of SA of pathology image analysis workflows, which leverages the use of large-scale distributed environments, as further described in Section 3. This system also includes a number of commonly used SA methods, a custom multilayered storage system and provides graphical tools to simplify its use.
- We propose computation reuse algorithms that exploit multiple levels of reuse (coarse-grain and fine-grain). The algorithms developed and evaluated include: Smart-Cut Algorithm (SCA), Reuse-Tree Merging Algorithm (RTMA), and TRTMA (see Section 4). The computation time reductions observed in the experimental results with the proposed strategies as compared to not reusing computation are of up to 2.9×. Furthermore, the fine-grain reuse improved the performance of the coarse-grain only case in about 1.5×.
- We experimentally evaluated the motivating application[7] on hybrid machines equipped with CPU and Intel Phi. As presented in the experimental results, the cooperative use of these processors with adequate scheduling strategies improved the application performance in about 2×, as compared to the CPU only execution. We executed large-scale studies on hybrid machines with 256 nodes (a total of 4096 CPU cores and 256 Intel Phis), in which a parallel efficiency of over 90% has been achieved.

The aggregate improvement attained with the proposed optimizations and scalability of the solution should enable SA in large-scale studies. These studies have the potential to significantly help the development of more robust applications, which are critical in the biomedical domain. The remainder of this paper is organized as follows: Section 2 describes the motivating application and the Region Templates (RT)[16] infrastructure in which our solutions were built. The overview of the proposed strategy to perform SA in the pathology image analysis domain is described in Section 3. The optimizations for computation reuse are detailed in Section 4. Further, we carry out the experimental evaluation in Section 5, describe the related work in Section 6, and conclude the paper and present the future directions in Section 7.

2 BACKGROUND

This section details the motivating pathology image analysis application workflow used in our SA studies (Section 2.1) and presents the Region Templates system in which our solutions were built for efficient and scalable execution (Section 2.2).

2.1 Motivating application

The use of modern high-resolution whole slide scanners is transforming the pathology image analysis domain. Their capacity of quickly acquiring high-resolution slides has motivated several recent works on tissue biomarker and image-based diagnostics.[7,9,17] These scanners capture very large 2D color images with resolutions of over 100K×100K pixels with about 50 GB in size or z-stacked images with several channels. Technology improvements, as slide loaders, allowed for several scans to be performed in a day, and as a consequence, multiple large private and public repositories of WSIs such as The Cancer Genome Atlas (TCGA) that contains over 30 000 such images were developed. In this context, the human-based data analysis may be inefficient due to the large amount of data available, the known subjective assessment of the data, the need for reproducing analysis, etc. Thus, the support and development of automated analysis tools are becoming a critical aspect for the progress of the domain.

A pathology image analysis workflow may consist of several computing steps (or stages), but Normalization, Segmentation, and Feature Extraction are typically employed on this class of applications. The development of efficient implementations of workflows with these stages has been the focus of our research in the recent years.[14,16] The Normalization is executed with the goal of reducing differences between images due to mechanisms in the acquisition process, which may lead, for instance, to different color intensities. The Segmentation, on the other hand, identifies objects of interest and delineates their boundaries. In our case, these objects consist of cells' nuclei. The Feature Extraction computes a vector of characteristics on a per object basis containing information that includes texture, shape, etc.

The motivating image analysis workflow used in this work is presented in Figure 1, which shows the decomposition of the workflow stages into their fine-grain internal operations. The operations in these stages have been developed, targeting CPU and Intel Phi for cooperative execution on hybrid systems.[18] One of the main concerns with the use of

automated microscopy image analysis refers to the robustness of the outputted information or results. This occurs because these application workflows are typically parameterized, and changes in the input parameters values may significantly affect the output results. For the segmentation stage, the parameters are presented in Table 1, along with the possible values that each parameter may assume, as suggested by an application expert. The number of parameters is high, making it very challenging and error-prone to vary them manually to identify changes in the output (eg, segmentation results). This creates a demand for using systematic methods to perform the parameter studies. It is important to highlight that this work has focused on studying parameters of the Segmentation stage of the application. This has been our choice because the Segmentation is the most parameterized stage, and it also extracts critical information (objects boundaries) used in the remaining of the downstream analyses.

The internal operations used in each of the computation stages presented in Figure 1 are detailed in Table 2. We have tried to employ existing works/tools in the implementation of the operations to assert their efficiency and implemented the ones not found in the literature (marked as “Ours” in the “Source code ref.” table column). The operations in the Normalization stage are typically data (pixel) parallel. The Segmentation stage has a few initial data parallel operations but uses irregular operations, for instance, based on flood-fill algorithms.[19] The Feature Extraction stage computes features based on objects identified in the Segmentation stage and, as such, employs object parallelism.

2.2 Region templates (RT)

This section describes the RT framework[16] used in this work as a baseline tool in which our proposals were built. RT supports the efficient execution of workflow applications in a distributed memory hybrid setting. The RT workflows are described in a hierarchical manner, such that a coarse-grain workflow formed of computation stages may have each of the stages implemented using another workflow of fine-grain tasks. This representation raises opportunities in terms of scheduling for efficient use of hybrid machines (eg, equipped with CPUs and accelerators), since heterogeneity in the performance of tasks at a fine granularity is exposed to the runtime.

An overview of the Region Templates architecture and execution scheme is presented in Figure 2. The RT is developed on top of the following core components: the runtime system, the data storage layer, and the data abstraction. The runtime system includes scheduling strategies for dispatching the execution of application stages and tasks. In RT, application stages are instantiated into the Manager process, which distributes them in a demand-driven fashion among the Worker processes. Each Worker is then responsible for the execution of stage instances assigned to it.

The execution of a stage instance within a Worker starts by reading input RT data objects, whereas its end is followed by a step in which RT objects produced/modified are pushed to the storage layer. As such, applications developed on top of RT do not have to handle communication among stage instances via traditional send/receive mechanisms. Instead, communication is performed by reading/writing RT objects from/to the storage. This approach alleviates the application development effort and also allows for the runtime

system to control the data placement, enabling the implementation of transparent data-aware scheduling of stage instances. The Region Templates data abstraction or data objects supported consist of data structures found in applications that compute data elements represented in low-dimensional spaces (1D, 2D, or 3D spaces) with or without a temporal dimension. Some of the data structures/elements used are: pixels, points, arrays (for images and 3D volumes), and polygons to represent segmented and annotated objects.

Moreover, during the execution of a stage within a Worker, tasks can be processed using all computing elements available in the machine hosting that Worker (ie, CPU cores or accelerators). This design creates a single Worker process per compute node on distributed memory machines, reducing process management overheads as fine-grain tasks executed by a stage instance are scheduled locally by Workers. The fine-grain scheduling is important to exploit the heterogeneity among the performance of tasks in hybrid systems. In real-world complex applications, it is expected that several operations will be used, and the different computation and data access patterns of these operations will make them benefit differently from the available processors. As such, not all tasks/operations will have the same speedups when executed on an accelerator. Thus, taking these variations into consideration can lead to a more efficient use of the system's available resources. In order to optimize the execution of fine-grain tasks in each Worker, we have employed Performance Aware Task Scheduling (PATS) strategies proposed in our previous work.[18,24] With PATS, tasks are assigned to either a CPU core or an accelerator based on the estimated acceleration of the task to each device and the current devices load, prioritizing the execution of a task in the processor in which it attains the highest speedup. For this sake, we maintain a list of tasks ordered by their expected speedups for the accelerator in each Worker and select those with highest and smallest speedups, respectively, for execution in the accelerator and the CPU.

Several other runtime systems have developed scheduling approaches for efficient use of hybrid machines.[25–29] However, most of the solutions focus on applications whose internal tasks have similar performance (speedups) when executed on an accelerator, as compared to the execution using a CPU. The PATS strategy, as described, considers variability in the tasks performance to better use the available processors. In our earlier work,[16] we evaluated PATS using a two-stage application analysis workflow consisting of segmentation and feature computation stages. In this work, the application analysis workflow consists of normalization, segmentation, and comparison stages. We target the execution of this workflow for algorithm SA and compare the scheduling to time-based approaches.

3 THE FRAMEWORK FOR EFFICIENT PARAMETER SENSITIVITY ANALYSIS

This section describes our framework for executing parameter studies in microscopy image analysis, whereas the optimizations targeting this class of applications are described in the next section. The overall design of our framework is presented in Figure 3. The system is built from several building blocks: the Region Templates runtime system, graphical interfaces for facilitating the application deployment, code generation tools, spatial indexing

for comparison of segmentation results, and methods for computing sensitivity analysis (SA).

In order to perform a parameter sensitivity analysis study, the system receives as input the imaging dataset, a representation of the application workflow described using the graphical interface, a list of parameters to be varied along with their value ranges, the SA method to be employed, and a comparison metric chosen by the user to evaluate changes in the application output. The SA method then generates parameter sets (parameter sampling) that should be used in the application execution. These parameters are passed through a computation reuse analysis (described in Section 4) to eliminate redundant processing, and after that, the application is instantiated for execution on top of the Region Templates (RT) environment. The RT executes the application in a distributed memory machine and manages the input, output, and intermediary data used/generated by the application stages.

After the actual segmentation is computed and objects of interest (cells' nuclei) are identified, such objects are compared to a set of reference objects computed using the application default parameters. In this phase, objects are loaded in efficient indexing structures (eg, R-trees[30]) to speedup this phase by avoiding, for instance, each object found in the segmented image to be compared to each other in the set of reference objects. All application stages, including the indexing, may have multiple instances in the distributed environment to allow for a scalable execution. Finally, the comparison phase is finalized with an output metric value (Dice, Jaccard, etc) that quantifies the variation in the segmentation results computed with a given parameter set as compared to the reference data. This information is inputted to the SA methods, which then output the correlations among input parameters and the segmentation output.

The sensitivity analysis may be carried out using a single method or by combining multiple methods. The main SA methods supported in our system are Morris One-At-A-Time (MOAT),[3] methods to compute importance measures as Pearson's and Spearman's correlation coefficients,[31] and the Variance-based Decomposition (VBD) method.[4] The MOAT computes global sensitivity analysis by varying one input parameter per time, resulting in a number of r changes for each studied parameter with r typically in the range of 5 to 15. This method is less compute demanding (smaller number of parameter samples/evaluations need) as compared to the other ones, but it is also less informative and is a good candidate to be used in the beginning of studies, for instance, to filter out unimportant parameters before other methods are applied. The methods that compute importance measures, eg, Pearson's correlation coefficient (CC) and partial correlation coefficients (PCC), vary multiple input parameter values at a time in order to identify non-linear effects, [31] but they are more demanding with respect to the parameter sampling size. Furthermore, the most demanding approach is VBD. It is able to split uncertainty in the output among parameters and can also account for non-linear relationships among them. It requires a large number of application runs, with the per parameter sample size in the order of hundreds to thousands. Regardless of the SA method employed, the user can choose different techniques to build the parameter sets. The ones we support include: Monte Carlo sampling, Latin hyper-cube sampling (LHS),[32] and quasi-Monte Carlo sampling with Halton or Hammersley sequences. These strategies are known to cover or explore well the parameter space. Table 3

shows an example output for both the MOAT and VBD methods applied to our motivating application. In this example, we have first executed MOAT to reduce the number of parameters studied by VBD.The VDB results show, for instance, that *G*2 explains alone (main effect) 73% of the output variation, whereas parameters such as *T*2, *MinSize*, *MinSizePl*, and *Watershed* have very small effect on the output. The total value includes also the impact interactions with other parameters.

We have also developed a graphical interface to simplify the development of the analysis workflows on top of our system and make it more accessible for non-experts in high-performance computing. Our tool uses the workflow model descriptions and tools available in the Taverna Workbench,[33] extending them to allow for hierarchical workflows and to include parameters to be evaluated in a SA study.

4 MULTILEVEL COMPUTATION REUSE

This section describes the proposed computation reuse techniques. In SA studies, the application is executed multiple times over the same dataset as input parameter values are varied. However, the multiple parameter sets generated for execution may have subsets of values that are common, which raises opportunities to reuse the computation. A common computation is defined here as a set of deterministic stages or tasks that have the same input data and parameter values, thus resulting in the same output. This is illustrated in Figure 4 that exemplifies two schemes for instantiating the application workflow for execution of a SA. The first creates a copy of the entire workflow for each parameter set, whereas the second shows a compact workflow execution with computation reuse.

As discussed, the opportunities for reuse in our hierarchical workflows occur at both stage and task levels. The reuse of stages can be attained through the removal of repeated/common stage instances with the correct routing of output from the remaining stages and changes in dependencies. The reuse level can be improved by searching for partial stage (tasks) reuse opportunities. For instance, if only a subset of the parameters is common among stage instances, the reuse will not be viable at the stage level, but parts of the computation carried out by the stage's internal tasks may still be reused. In order to do it, stage instances with partial reuse opportunities can be merged together into a single, more coarse-grained, stage instance to have the reusable tasks executed only once.

Reuse at the task level is, however, more challenging since the stage instance resulting from the merging of multiple instances is executed within a single node. Thus, the merging algorithm needs to take into account that (i) the merging may generate a stage with higher resource demands (eg, memory requirement) and (ii) the merging reduces the parallelism available as the number of stage instances is decreased. In this case, given that a limited number of stage instances can be merged together, an efficient merging algorithm should search for the merging choice that optimizes the amount of reuse. The algorithm should also take into account the impact of the merging to the parallelism. Our strategies for computation reuse are described in the remaining of this section.

4.1 Stage-level computation reuse

The computation reuse at the level of stages is performed by identifying stage instances that use the same input data and parameter set. In cases in which it occurs, the duplicate stage instance is removed, and a compact workflow is built (Figure 4). This process is presented in Algorithm 1. The algorithm receives as input the application workflow (*appGraph*) and the set of parameters (*parSets*) to be evaluated. The compact workflow (*comGraph*) is then created by merging to it instances of the workflow created for each of the parameter sets to be evaluated (lines 3–6).

The MergeGraph is responsible for adding each workflow instance to the compact graph representation. This is computed by simultaneously exploring the received workflow instance entry point (*appVer*) and the compact graph (*comVer*) to identify stages from *appVer* that are already in *comVer* and, as such, should not be inserted again in the compact representation. This is computed in the main loop of the procedure (lines 8–29), which will find each child node of *appVer* and check whether it is also available in *comGraph* (line 9). If this is the case, the merging procedure is called recursively to children nodes of v and v' or to the rest of the workflow on that branch. The find method will check the input data, stage name, and input parameter values to compute the matching.

When a corresponding vertex of *appVer* has not been found in *comVer.children*, two cases should be taken into consideration (lines 11–28). The first is the most obvious configuration in which the node from *appVer* has not been added to *comVer*, and as such, it is created and added to the compact graph (lines 13–19). To verify if this is the case, the algorithm needs to assert that the node v being added has not already been included in the graph in the merging of another path of the workflow. This could occur for nodes with multiple dependencies as it did with D from the example presented in Figure 4. If the entire path A, B, and D is added to *comVer*, when processing C, D should not be added again. When this case is identified (lines 21–26), only the dependencies are correctly set. The *PendingVer* look-up table is used to store and identify those stage instances with multiple dependencies. Before inserting a stage instance, *PendingVer* is first consulted to check whether that stage was not yet created (line 12).

Algorithm 1

compact Graph Construction Algotithm

```
1:  Input: appGraph; parSets;
2:  Output: comGraph;
3:  for set ∈ parSets do
4:      appGraphInst ← instantiateAppGraph(set);
5:      MergeGraph(appGraphInst.root, comGraph.root);
6:  end for
7:  function MERGEGRAPH(appVer, comVer)
8:      for v ∈ appVer.children do
9:          if (v′ ← find(v, comVer.children)) then
```

```
10:          MergeGraph (v, v′);
11:        else
12:          if ((v′ ← find(v, PendingVer))==∅) then
13:            v′ ← clone(v)
14:            v′.depsSolved ← 1
15:            insert(v′, comVer.children)
16:            if |v′.deps| ≥ 1 then
17:              insert (v′, PendingVer)
18:            end if
19:            MergeGraph(v, v′);
20:          else
21:            insert(v′, comVer.children)
22:            v′.depsSolved ← v′.depsSolved+1
23:            if v′.depsSolved == v′.deps then
24:              removed (v′, PendingVer)
25:            end if
26:            MergeGraph(v, v′)
27:          end if
28:        end if
29:      end for
30:    end function
```

As presented, for each instance of the application workflow (*appGraphInst*) using different parameter values, the k stage instances of that workflow need to be merged into the compact graph. Each of the k stage instances is merged via a MergeGraph call, which has the complexity dominated by the *find* in *comVer.children* that has up to n parameters sets elements (see Figure 4, node A on the compact composition: 1 child for each parameter set). However, by employing a hash table to compute the find operation, the complexity is $\mathcal{O}(1)$. The n workflow instances (one for each parameter set) will insert k stage instances each for an overall cost of $\mathcal{O}(kn)$.

4.2 Task-level computation reuse

This level of reuse allows for partial inter-stage (tasks) reuse when only a subset of parameters used by stage instances match. In the proposed strategies, stages with partial reuse are merged into a single more coarse-grain stage, and common internal tasks of those stages are reused to avoid duplicate computations. An obvious solution to the problem would be maximizing reuse and merge all stage instances with any possible level of reuse. This strategy, however, may result into very coarse-grain stages, which would require more resources than those available in a node (Worker). Moreover, it may substantially reduce parallelism (number of stage instances) even when the reuse gains are small.

In order to address the first issue, we have redefined the reuse problem by creating a limit to the number of stage instances that are merged together (*MaxBucketSize*). This value can be chosen, for instance, to ensure that merged stages will not oversubscribe the resources or to

guarantee that a minimum number of stage instances will be available for parallel execution. In the rest of this section, we present four approaches to the problem.

4.2.1 Naïve merging algorithm—This approach is the simplest strategy. It iterates through the list of application parameters for each stage and group subsequent stages instances into buckets of size *MaxBucketSize*. A bucket holds stage instances that are merged together. As may be noticed, this strategy relies on the order that parameter sets are inputted, which limits its reuse performance. This algorithm has a complexity $\mathcal{O}(n)$ in which n is the number of parameter sets or workflow instances to be merged.

4.2.2 Smart cut algorithm (SCA)—The second approach to the problem uses a graph to represent stage instances and the benefits (computation reuse amount) due to their merging. The representation employed is a fully connected undirected graph in which the stage instances are the nodes and each edge is the degree of reuse between two stage instances (see Figure 5B). The degree of reuse is defined as the number of tasks that would be reused by merging a pair of stage instances. After that, we define the stage instances to be merged by partitioning the graph in subgraphs until the subgraph has at most *MaxBucketSize* stage instances/nodes. This is carried out by employing successive min-cut[34] operations in the graph.

Our approach recursively employs the 2-min-cut algorithm (ie, cuts the graph in two subgraphs, minimizing the sum of the weights of edges crossing the cut[34,35]) as illustrated in Figure 5. Given the input parameters and intra-stage workflow of tasks, the fully connected graph in Figure 5B is created. The first 2-min-cut is then performed (Figure 5C) to remove *c*, which is the stage instance least related to the subgraph with the remaining nodes. The 2-min-cut is then applied again to the largest subgraph, removing nodes *a* and *b* from the main subgraph. This process is repeated until a bucket of size 2 (MaxBucketSize=2) or smaller is reached (see Figures 5C and 5D). Those nodes selected to be merged together (*d* and *e*) are removed from the originally graph, and the process is repeated with the remaining stages until all stages are grouped into buckets.

This procedure is presented in Algorithm 2. SCA starts with a graph containing all stages (*stages*) and iterates through the cutting process until all stages are assigned to the *bucketList* (lines 3–14). Successive 2-min-cut operations are performed on the current graph (*stages*), which divides it into two disconnected subgraphs (*s*1 and *s*2 in lines 4 and 7). This is done until the largest subgraph resulting from a cut (lines 5 and 8) does not fit into a bucket (ie, contains over *MaxBucketSize* stage instances, as shown in line 6). When it fits, the largest subgraph *s*1 is defined as a bucket and added to the output list of buckets (line 10). Those nodes (or stage instances) are then removed from the original graph (lines 11–13). The entire process is repeated until all stage instances are assigned to a bucket.

Algorithm 2

smart Cut Algorithm(SCA)

1: **Input:** stages; MaxBucketSize;

```
2:  Output: bucketList;
3:  while |stages| > 0 do
4:    {s1,s2} ← 2minCut(stages)
5:    s1 ← max(s1, s2)
6:    while |s1| > MaxBucketSize do
7:      {s1,s2} ← 2minCut(s1)
8:      s1 ← max(s1, s2)
9:    end while
10:   insert(s1, bucketList)
11:   for each s ∈ s1 do
12:     remove(s, stages)
13:   end for
14: end while
15: return solution
```

In the worst case, if n stage instances are available for merging, the 2-min-cut is applied $\mathcal{O}(n)$ times for each bucket created, and about $n/MaxBucketSize$ buckets would be generated. This results in total of $\mathcal{O}(n^2)$ 2-min-cuts. The implementation of the 2-min-cut used in our work employs a Fibonacci heap[34] with a per-cut complexity of $\mathcal{O}(E + V log V)$. Since the graph is fully connected, a cut will cost $\mathcal{O}(n^2)$. Therefore, the entire SCA is $\mathcal{O}(n^4)$. Although interesting, as presented in Section 5, the SCA is not useful in practice in several scenarios because of the high complexity/execution time.

4.2.3 Reuse-tree merging algorithm (RTMA)

The RTMA describes and organizes stage instances as a tree structure. In order to do that in a way reuse can be identified, we proposed a novel tree structure called reuse-tree. In this tree, stage instances are organized according to the parameter used and, as a consequence, by their internal workflow of tasks. In essence, stage instances with common tasks share parents on the tree, and each level of tree represents a parameterized task. Stage instances having tasks with same parameter values are stored into the same branch of the tree.

An example reuse-tree is presented in Figure 6B. In this example, the application stage is implemented using three tasks organized into a pipeline. During the insertion of x, for instance, starting from the root node (black node), it is checked if another task t_1 has already been inserted using the same parameter value (annotated in each edge) (Figure 6C). In our example, t_1 with $p_1 = 8$ already exists. Thus, we follow that path in tree and search for the next parameter (or multiple parameters if the tasks use more than one) on the right subtree (Figure 6D). Since node's 2 only child (node 5) cannot be reused for stage instance x, because the second parameter value p_2 of x differs from the one used in node 5, a new node representing this non-reusable task is created (node 6) as shown in Figure 6E. Finally, since node 6 is new, there cannot be any more reuse opportunities from that point forward. Thereby, a single child node must be created for each of the remaining tasks (Figure 6F).

The algorithm implemented on top of the reuse-tree merges stage instances in buckets of *MaxBucketSize* elements by traversing the tree in an bottom-up fashion. This process is illustrated in Figure 7 and Algorithm 3, which starts with a reuse-tree with 12 stage instances ($S1$ to $S12$) and employs a *MaxBucketSize* = 3. On Algorithm 3, parents of leaf nodes are selected and have their children grouped in buckets of exactly *MaxBucketSize* elements (lines 6 and 7). Thus, if a parent has less than *MaxBucketSize* children, nothing is done. However, if more than *MaxBucketSize* nodes exist and they are not multiple of *MaxBucketSize*, the remaining nodes that do not fill a bucket are kept in the tree. Examples of these cases can be seen in the transition from Figure 7B to Figure 7C, where, for instance, the buckets ($S1$, $S2$, and $S3$) and ($S4$, $S5$, and $S6$) are created and $S7$ is kept in the tree.

This process continues by removing merged nodes from the tree. If a parent node ends up grouping all of its children in buckets, it must be removed from the tree (node 5 on Figure 7C). This process is performed recursively by removing the childless parent nodes and then checking if the removal of a node makes its parent childless. The final step of the merging is to move the leaf nodes up one level in order to enable the creation of new buckets. The operation *MoveReuseTreeUp* (Algorithm 3, line 9) that moves remaining leaves to their parents' ancestors (eg, nodes $S7$, $S8$ and $S9$ of Figure 7D are placed as children of node 2 on Figure 7E). Further, the same merging process is re-executed until the tree height becomes 1. The rest of the nodes are then added into new buckets of size not larger than MaxBucketSize (lines 11–14).

Algorithm 3

Reuse-Tree Merging Algorithm(RTMA)

```
1:  Input: stages; maxBucketSize;
2:  Output: bucketList;
3:  bucketList ← ∅;
4:  rTree ← GENERATEREUSETREE(stages)
5:  while rTree.height > 2 do
6:      leafsPList ← GenerateLeafsParentList(rTree)
7:      newBuckets ← PruneLeafLevel(rTree, leafsPList, maxBucketSize)
8:      insert(newBuckets, bucketList)
9:      MoveReuseTreeUp(reuseTree, leafsPList)
10: end while
11: if rTree.rootchildren ≠ ∅ then
12:     newBuckets ← rTree.root.children
13:     insert(newBuckets, bucketList)
14: end if
15: return bucketList
```

In RTMA, the tree is created by adding n stage instances with k tasks each (GENERATEREUSETREE). Each task insertion may look for tasks with the same parameter values in that level (using a look-up table) in $\mathcal{O}(1)$, which results in $\mathcal{O}(k)$ cost for inserting each stage instance. Given that n stage instances are inserted, this phase is $\mathcal{O}(kn)$. Further, the

actual merging is performed in lines 5–10 of Algorithm 3. The GenerateLeafsParentList is $\mathcal{O}(kn)$ per iteration in the worst case, with $k-1$ iterations. As such, the entire cost of all iterations is $\mathcal{O}\left(k^2n\right)$. The pruning (PruneLeafLevel) cost is constant per node added to a bucket; thus, its complexity is $\mathcal{O}(n)$ for all iterations of the loop. The MoveReuseTreeUp worst case takes place when no reuse exists. In this case, all leaf nodes are moved up $k-1$ times, resulting in a complexity of $\mathcal{O}(kn)$. As such, the algorithm's execution time complexity is $\mathcal{O}\left(k^2n\right)$ Because k is a fixed value per stage and $n \gg k$, the algorithm is close to linear in practice. In the case studied in this work, the segmentation workflow has $k = 7$ tasks only and n ranges from 160 to 10 000 according to the SA method used.

4.2.4 Task-balanced reuse-tree merging algorithm (TRTMA)

As previously discussed, stage merging may impact the application scalability by substantially reducing the ratio of stage instances available by computing cores used. This scenario is likely to occur when the computing environment is large and/or a small to moderate parameter sample size is used. On these cases, two problems may emerge: (i) merging could lead to load imbalance among nodes because some stage instances may be more costly and (ii) the number of stage instances (after merging) could be insufficient to use all nodes/computing cores available. The TRTMA algorithm proposed in this section addresses these aspects.

TRTMA performs merging while attempting to balance the buckets' cost, which, in our case, are estimated by the number of tasks to be executed in that bucket after merging. As such, TRTMA considers the compromise between merging stages and the potential imbalance it could cause, performing this analysis on top of a reuse-tree. The algorithm is implemented into three phases: Full-Merge, Fold-Merge, and Balance. The first two perform an initial computation of the buckets (stages to be merged), whereas the last phase minimizes imbalance or difference in cost among buckets.

The Full-Merge traverses the reuse-tree on a top-down fashion, attempting to find a tree level in which there are at least *MaxBuckets* (the number of buckets that the algorithm will create) nodes. This process can be seen on Figure 8 for *MaxBuckets* = 3. The first level of the tree is visited and 4 nodes are found: 1, 2, 3, and 4 (see Figure 8B). In this case, a single bucket is created per node, each of them containing stage instances stored in leaf nodes on the subtrees rooted on nodes at that level (see Figure 8C). If the number of buckets created is greater than *MaxBuckets*, which is the case of our example, the Fold-Merge operation will be executed to combine buckets and reduce their number to *MaxBuckets*. Otherwise, if the number of buckets is already *MaxBuckets*, nothing is done in the Fold-Merge phase.

The Fold-Merge phase sorts the buckets according to their cost (number of tasks after merge, ie, the number of nodes of the subtree) and combines buckets with the smaller costs to the buckets with higher costs. This process is presented in Figure 9, where after the folding pivot point the buckets are merged. In the example, the folding has resulted in merging buckets pairs $<b_4, b_5>$ and $<b_3, b_6>$. This merging algorithm is an attempt to create an initial solution with the exact number of required buckets (*MaxBuckets*) while reducing the imbalance. If the number of buckets after Full-Merge is greater than 2 × *MaxBuckets*, the

Fold-Merge is executed multiple times until the *MaxBuckets* number of buckets is reached. In the example of Figure 8, the initial 4 buckets of Figure 8C have costs 5, 9, 5, and 3, which results in the 3 buckets of Figure 8D with costs 8, 9, and 5, after the Fold-Merge.

The Balance phase of our algorithm will try to move stage instances among buckets previously generated with the goal of minimizing the buckets imbalance. This is implemented in our algorithm through improvement operations, which are repeatedly applied to pairs of buckets. The actual buckets used in each improvement operation are the least and most expensive ones, which are called here, respectively, *smallRT* and *bigRT*. The improvement operation will then move stage instances from *bigRT* to *smallRT* and check if there is a reduction in the overall imbalance between these buckets. In addition, it also checks if moving stages between the buckets has reduced the cost of the most expensive bucket (*bigRT*). The transfer of stage instances is only carried out or committed if both imbalance and maximum cost are reduced.

The Balance phase of the TRTMA is presented in Algorithm 4. The main loop (lines 3–12) identifies the most and least expensive buckets and tries to improve the current solution by moving stage instances from *bigRT* to *smallRT* with calls to the function SingleBalance. If a better solution is found, the tree node branch from *BigRT* that should be transferred to *smallRT* is returned (improvement), and all stage instances in that tree branch are moved to *smallRT* (line 8). In this case, *smallRT* and *bigRT* are modified and reinserted in the bucketList so that the list remains sorted (lines 9 and 10). If no improvement is returned, the algorithm ends and returns the bucketList (line 13) for execution.

Algorithm 4

The Balance step of the TRTM

```
1:  Input/Output: bucketList;
2:  bucketList is a sorted data structure by descending cost (e.g., multiset)
3:  Repeat
4:    bigRT ← firstElement(bucketList)
5:    smallRT ← lastElement(bucketList)
6:    improvement ← SingleBalance(bigRT, smalIRT)
7:    if improvement ≠ ∅ then
8:      TransferSubtree(improvement, bigRT, smalIRT)
9:      update(bigRT, bucketList)
10:     update(smallRT, bucketList)
11:   end if
12: until improvement ≠ ∅
13: return bucketList
4:  function SINGLEBALANCE(bigRT, smallRT)
15:   improvement ← ∅
16:   imbal ← | TaskCost(bigRT) − TaskCost(smallRT) |
17:   while (c ← NextDFSNode(bigRT)) ≠ ∅ and imbal ≠ 0 do
18:     newBigRT ← bigRT
```

```
19:        newSmallRT ← smallRT
20:        TransferSubtree(c, newBigRT, newSmallRT)
21:        newImbal ← | TaskCost(newBigRT) − TaskCost(newSmallRT) |
22:        newMakespan ← max(TaskCost(newBigRT), TaskCost(newSmallRT))
23:        if newImbal < imbal and newMakespan < TaskCost(bigRT) then
24:          improvement ← c
25:          imbal ← newImbal
26:        end if
27:      end while
28:      return improvement
29:    end function
```

The SingleBalance function receives two buckets as input and searches for stage instances on *bigRT* to be sent to *smallRT* in an attempt to improve the solution. It searches for improvements in a depth-first walk manner in BigRT (line 17) by evaluating all subtrees in *bigRT* and testing if their transfer to *smallRT* characterizes an improvement (lines 17–27). For each node *c*, new buckets (*newBigRT* and *newSmallRT*) are computed (line 20), and it is checked if the transfer of the subtree with root *c* from *newBigRT* to *newSmallRT* results in a better solution than the current one. For this sake, the algorithm evaluates if the new buckets have a smaller imbalance and if the maximum cost of the new buckets is smaller than the original (lines 19–21). In case a better solution is found, the improvement node and imbalance are updated, and the algorithm continues the search for better solutions. The algorithm then returns the least imbalanced improvement that also reduces the maximum cost as compared to the input buckets if one exists (line 28).

Figure 10 presents an overview of the balance phase along with examples of the improvement operation. First, the *bigRT* and *smallRT* buckets (see Figure 10B) are selected as the candidate pair for improvement. The Balance will try to send one of the subtrees of *bigRT* along with all stage instances it stores to *smallRT*. The first attempt of improvement is to move the subtree rotted at node 6 to *smallRT*, as shown in Figure 10C. However, if this is performed, the imbalance among those buckets would increase to 7, and thus, this operation is not executed. Further, it tries to move the subtree starting in node 7 to *smallRT*. In this case, the imbalance decreases from 4 to 3. As such, the next test is performed to check whether the maximum cost between *smallRT* and *bigRT* was reduced. Unfortunately, this is not the case because this movement would make *smallRT* to have 9 tasks, which was already the value of the original *bigRT*. As such, this operation may not benefit the application makespan. Finally, by applying the improvement of leaf-node $S9$, the resulting buckets would be {$S4$, $S5$, $S6$, $S7$, $S8$} and {$S9$, $S10$, $S11$}, both with cost 8 (see Figure 10E). It is worth noting that the selection of node $S9$ is only for the sake of presentation in Figure 10E, being this node interchangeable with nodes $S4$-$S8$ without impacting the algorithm outcome. Since this improvement operation also reduces the maximum cost as compared to the original configuration and is the best solution found, it is applied (see Figure 10F). In this example, the Balance operation will then finish since the new imbalance is 0.

Complexity: The TRTMA cost is dominated by the Balance step. This phase applies SINGLEBALANCE until the buckets cannot be further balanced. Thus, the TRTMA complexity can be derived from the number SINGLEBALANCE executions. For that sake, we rely on the Sum of Imbalances (SI), which is defined here as the sum of differences in imbalance between pairs of buckets in a list (BL) of size n sorted by their costs: $SI(BL) = \sum_{i=1}^{n/2} |BL_i| - |BL_{n-(i-1)}|$ The module of a bucket ($|BL_i|$) is defined as the cost of the bucket or the number of tasks it executes (eg, given a sorted bucket list with 4 buckets of costs 9, 5, 5 and 3, $SI(BL) = (9 - 3) + (5 - 5) = 6$).

A SINGLEBALANCE either reduces the SI, if an improvement is found, or terminates the TRTMA. In case of an improvement, SI is reduced by at least 2 in the worst case when a stage instance is moved between the two buckets (the costs of the moved stage on both buckets is 1 either because the stage have a single task or if the reuse is maximum on both buckets). Therefore, for a given SI, TRTMA finishes in at most $SI/2$ SINGLEBALANCE calls. The SINGLEBALANCE attempts to find improvements through TRANSFERSUBTREE calls (see Algorithm 4, line 20) that move the c subtree from *bigRT* to *smallRT*. The TRANSFERSUBTREE will insert each stage instance rooted at c in *smallRT* with a cost of k (fixed) for each instance. A subtree c on its turn may have $\mathcal{O}(n)$ nodes, being n the number of stage instances. Thus, each TRANSFERSUBTREE call is $\mathcal{O}(kn)$ in the worst case. As SINGLEBALANCE will try to move each of the tree nodes from *bigRT* to *smallRT*, and *bigRT* may have $\mathcal{O}(n)$ nodes, each SINGLEBALANCE call is $\mathcal{O}(kn^2)$ in the worst case. Further, the SI is bound by n, and the worst case of TRTMA is $\mathcal{O}(kn^3)$.

Discussion: In practice, we have observed that the worst case is unlikely to occur due to the aspects discussed below.

- The number of *MaxBuckets* employed is typically high and computed with respect to the number of computing cores available in the system. We have experimentally found the best value to be about 3× the number of computing cores. Thus, a large number of buckets is created in the first two phases of TRTMA, which, as a consequence, reduces the imbalance among buckets and the SI.
- The parameter sampling strategies employed are expected to sample the parameter space with minimal bias, which is especially true for quasi-Monte Carlo sampling strategies using low-discrepancy sequences.[36,37] As a consequence, the reuse trees built by the TRTMA are expected to contain a similar number of stages instances stored in all branches at each level of the reuse tree. If this occurs, buckets inputted to the *Balance* step will have a maximum imbalance between pairs of buckets smaller than $n/MaxBuckets$. This reduces the number of required *SingleBalance* calls.
- Multiple optimizations were implemented in SINGLEBALANCE and TRANSFERSUBTREE to improve their performance. For instance, during the walk through the *bigRT* decedents in SINGLEBALANCE, if a node c has only a single child, its descendant node does not need to be evaluated as possible improvement

since moving it results in the same solution as moving *c* to *smallRT*. Moreover, the performance of TRANSFERSUBTREE can be improved in case a node being move from *bigRT* has no more reuse with any node in *smallRT* in a given level during the insertion. In that case, the entire subtree branch with all nodes it stores can be moved from *bigRT* to *smallRT* in a single insertion/operation.

For sake of profiling the performance of TRTMA, we have executed it with different *MaxBuckets* and *n* values. As is presented in Section 5.4 (Figure 16), its cost is subquadratic in our experiments and smaller for higher *MaxBuckets* values that, as discussed above, reduce imbalance. In addition, we also want to highlight that as the TRTMA algorithm considers the MaxBuckets to compute the merging and reuse, and values of MaxBuckets can be setup considering the number of computing devices available (ie, CPU cores). Thus, we expect that TRTMA will work well for machines with different hardware configurations or number of computing cores.

5 EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of our optimizations for SA, which were implemented on top of the Region Templates for parallel execution. We first analyze the system scalability, the impact of the scheduling on hybrid systems and further discuss the gains with the computation reuse algorithms proposed in this work. For sake of the evaluation, we have employed an application workflow used in brain cancer studies[7] (Section 2.1). It consists of the Normalization, Segmentation, and Comparison stages. The comparison is carried out in our experiments as the Dice values of objects in the reference and computed segmentation results (masks). The input data (see Table 4) consists of Glioblastoma Multiforme (GBM) WSIs downloaded from The Cancer Genome Atlas (TCGA) repository, which were partitioned into 4K×4 K image tiles for parallel execution. Most of the experiments were conducted on TACC Stampede cluster. Each node has a dual-socket Intel Xeon E5–2680 processors, an Intel Xeon Phi SE10P or MIC (Many Integrated Core) co-processor, and 32GB RAM. Because Stampede was upgraded for an Intel Phi–only system during the development of this work, the experiments on Section 5.3, which evaluate our proposed TRTMA algorithm, were executed on the PSC Bridges cluster. Each node on Bridges is comprised of two Intel Xeon E5–2695 v3 with 128GB of RAM. The application and middleware codes were compiled using Intel Compiler 13.1 with “-O3” flag. The experiments were replicated five times and claims for equivalence or difference between results in this paper are made based on t-test (two-tailed) with $P < 0.001$. The main dataset used in our evaluations is presented in Table 4, but subsets of the data are used in some of the experiments are described in each of the following sections.

5.1 Benefits of the cooperative execution using CPUs and Intel Phi

This section evaluates the application performance and scalability in distributed memory settings equipped with CPU and Intel Phi. The evaluation was carried out using the motivating application described as a hierarchical workflow. The tasks inside the Normalization and Segmentation are presented in Section 2.1. Further, the following application versions were used: CPU-only that employs all CPU cores available in the machine, MIC-only that employs the Intel Phi coprocessors in the execution, and CPU-MIC

that uses the CPU cores and Intel Phi in cooperation and multiple scheduling strategies to distribute tasks among processors in each node: First-Come, First-Served (FCFS), Heterogeneous Earliest Finish Time (HEFT), and PATS. The experiments in this section have been carried out using a weak scaling setting in which the number of computing nodes and dataset increase proportionately. For the configuration with 256 nodes, a dataset with 136 568 4K×4 K image tiles (6.5 TB of uncompressed data) was used. In the cases on which Phi is used, its number is equal to that of nodes. The experimental results are presented in Figure 11.

The application scaled well regardless of the configuration or version. Moreover, the cooperative execution of the CPU and Intel Phi (CPU-MIC) improved the performance of a single processor in all cases. However, the appropriate workload division among devices is significant as, for instance, the performance gains of PATS on top of FCFS are about 1.32× on average. As compared to HEFT, which is known to be an efficient scheduling strategy for hybrid systems, PATS is still 1.2× faster. The superior performance of PATS is a consequence of its capacity of taking into account that particular tasks may be more appropriate for different devices.

5.2 Effect of multilevel computation reuse for different SA methods and sampling strategies

In this section, we evaluate the impact of the proposed computation reuse algorithms for the MOAT and VBD SA methods. MOAT is used with all application parameters to identify the non-influential ones, and VBD is further applied with the remaining parameters. This particular set of experiments was executed using only 16 nodes because it intended to evaluate only the gains of the reuse optimizations. Experiments with a larger number of nodes with reuse strategies are presented in Section 5.3.

The reuse algorithms were first executed with MOAT using a *MaxBucketSize* of 7 and parameter sample sizes varying from 160 to 640. The *MaxBucketSize* of 7 was experimentally detected as an optimal value in this setup. The parameter sets were created with a quasi-Monte Carlo sampling using a Halton sequence. The execution times measured refer to the application makespan. The cost of performing the reuse analysis is highlighted in the upper part of the graphs bars. Five configuration were evaluated: “No reuse” that employs the replica based scheme, the “Stage level” that reuses only stage instances with the same parameters, and the “Task Level” that reuses fine-grain tasks and is executed with the Naïve, SCA, and RTMA algorithms. The TRTMA was not included on the results because it attains the same performance as RTMA in small scale settings.

The performance results are presented in Figure 12. As shown, the gains with reuse were significant in all configurations as compared to the “No reuse” case. The “Stage Level” was up to 1.8× faster than not reusing computation, whereas the “Task Level - Naïve” improves on the “Stage Level” 1.08×; this gain is significant according to the t test. The task level reuse with SCA and RTMA have, respectively, improved the “Stage Level” reuse in up to 1.39× and 1.5×. It is also possible to notice that the performance gains of RTMA are higher with larger sample sizes. The SCA, on the other hand, suffers from its high computational complexity as the sample size is increased and spends a significant amount of time

computing the reuse. This offsets the gains with reuse in this approach. Finally, in the best case, RTMA is about 2.6× faster than not performing reuse.

We have also analyzed the same set of algorithms in a VBD SA experiment. The VBD used the eight most significant parameters detected in the MOAT SA (Table 3). The main difference between MOAT and VBD in this evaluation regards the number of parameter sets executed. VBD demands hundreds to thousands runs per parameter; thus, we varied the parameter sample size or the number of parameter sets from 2000 to 10 000. To speedup this experiment, the number of nodes used was increased to 32. The results are shown in Figure 13. The gains with the computation reuse are also significant in this case. However, as expected, because of the larger number of parameter sets used, the SCA method could not even finish the reuse analysis within 14 000 secs. The RTMA had speedups of at most 2.9× against the “No Reuse” approach and 1.51× on top of “Stage Level.”

Further, we have varied the parameter sampling strategy used to generate the parameter sets during a SA analysis to measure the reuse opportunities with different strategies. In this experiment, the Latin hyper-cube sampling (LHS), Monte Carlo (MC), and quasi–Monte Carlo using a Halton sequence (QMC) methods were analyzed. In this experiment, we calculate the maximum reuse available at the fine-grain tasks, meaning that coarse-grain reuse has been performed before this analysis. It is measured in number of tasks that could be reused. The results are presented in Table 5. In essence, there is a small variability in the amount of reuse across sampling strategies or as the parameter sample set increases. These results are interesting as they show that the reuse is not limited to the use of a specific parameter sampling method/sample size.

5.3 The effect of the merging to scalability

This section evaluates the impact of merging to the scalability of the application in SA studies. As discussed before, the merging reduces the number of stage instances and, consequently, the parallelism available for execution on distributed memory systems. In order to evaluate this aspect, we have used a MOAT SA with a parameter sample size of 1000 and a VBD SA with 10 000 runs. For both studies, the number of nodes is varied to evaluate the performance. In essence, the MOAT study will represent a case in which the parallelism may not be high enough to fully utilize the system as the number of nodes used increases, whereas VBD is a very demanding study in which the parallelism available (number of stage instances) will continue to be sufficient to fully exploit the target machine even after stage merging. In these experiments, four versions of the application were executed: No reuse as a baseline, Stage Level reuse only, and fine-grain computation reuse with RTMA and TRTMA.

The experimental results for the MOAT are presented in Figure 14. As can be observed, the Stage Level reuse can attain good scalability as the number of nodes increases, which is a result of high level of parallelism available. The RTMA was set up to use MaxBucketSize of 10 (best value for 8 machines). The TRTMA, on the other hand, was configured to create a number of buckets of 3× the number cores available. The results show that RTMA and TRTMA have better performance than Stage Level only reuse until 32 machines are employed. After that, the performance of RTMA degrades as compared to the Stage Level

and TRTMA versions of the application. When 256 machines are used, TRTMA is close to 2× faster than RTMA and has a better performance than the Stage Level reuse. The RTMA performance degradation is a result of the low number of stages per core after merging. For TRTMA, this value is kept fixed (3× the number cores) for all configurations. As such, TRTMA can benefit of reuse in case of sufficient parallelism, similarly to RTMA, but can handle cases in which the parallelism may affect the performance.

Figure 15 presents the SA study using VBD with 10 000 runs. Here, as expected, the RTMA performance degradation never occurs because the number of stage instances available is high even after merging. Thus, the RTMA scalability is not affected and it attains speedups between 1.26× and 1.35× as compared with Stage Level only reuse. When comparing RTMA and TRTMA, their performances are equivalent, showing that TRTMA improves RTMA in cases of insufficient parallelism and does not degrade the performance when enough parallelism is available.

5.4 The performance of TRTMA as the parameter sample size and *MaxBuckets* are varied

This section evaluates the TRTMA execution time as the number of parameter sets (n) and the *MaxBuckets* are varied. As previously described in Section 4.2.4, the complexity of TRTMA in the worst case is cubic, but several optimizations were implemented to avoid this case. Thus, we wanted to evaluate its performance in practice. The execution times spent by TRTMA to perform the merging are presented in Figure 16. As presented, the execution time increases with n and reduces as *MaxBuckets* increases. However, the execution time growth with n observed is subquadratic in practice. Moreover, the reduction as the number of *MaxBuckets* increases was also expected because it reduces the imbalance of the initial solution used by the *Balance* step of the TRTMA.

6 RELATED WORK

The computation reuse, also known as value locality,[38,39] has been employed in several domains[40–42] using multiple techniques as value prediction,[43] dynamic instruction reuse,[44] and memoization.[45] We summarized the main related work and their features in Table 6. The reuse may be implemented in hardware using specific components specialized for this task or in software platforms. Further, the works are categorized here according to the approach employed: memoization and analytic. Memoization uses a cache-based solution to save/read results from previous computations. The analytic approach identifies reuse in the application, linking the output of operations to places in which it is reused to minimize or avoid caching. The next aspect is the granularity of the reuse. We have classified this aspect into two classes: Fine-Grain (instruction and compute inexpensive/small subroutines) and Coarse-grain (expensive routines or collections of subroutines and full applications). We differentiate fine-/coarse-grain because we leverage both levels in our approach; however, it is not explicitly distinguished in most of the related works. We have also analyzed if the solutions deal with large-scale datasets. Finally, the scope of reuse differs according to the applicability of the strategies to local or distributed computing environments.

Sodani and Sohi[44] employ a computation *reuse buffer* to optimize instructions execution with a memory cache. Their approach aims to reduce computational cost through reuse by

(i) ending the instruction pipeline earlier, thus also reducing resource conflicts, and (ii) by breaking dependencies of next instructions, which can be executed earlier since the necessary inputs are already available. The reuse buffer was initially proposed as a strategy to reduce branch misdirection penalties, but it was extended to other types of instructions. This work has been implemented in a superscalar pipeline, and has the limitation of being able of employing only small buffers in practice.

The work on by Richardson[45] deals with the fast execution of the called trivial computations, which occur for instance with a division by two that result in redundant computations and lead to opportunities of reuse. With respect to reuse, it quantifies the amount of redundant computation in several popular benchmarks, showing the benefits of memoization on those cases. Although this work focuses on quantifying the potential gains of reuse, it is performed on the scope of a hardware based solution. Wang and Raghunathan[40] attempt to reduce the energy consumption in embedded devices through a profiling-based reuse technique with memoization. The profiling identifies computation reuse regions and quantifies the benefits at different reuse granularities. Interesting discussions on the appropriate granularity of tasks are presented along with the limitations of a hardware based solution. In the works of Alvarez et al[46] and Modarressi et al,[47] a domain-specific strategy is proposed for reusing floating-point operations when operands are similar enough. The similarity metric in this case is a key aspect that affects reuse opportunities. As such, there is a trade-off between the precision of the computations and the reuse potential.

The work by Riera et al[48] leverages the error tolerance of deep neural networks (DNN) on a hardware implementation of a reuse-based DNN accelerator. By using the proposed accelerator, an improvement on energy utilization was observed. Since this approach for computation reuse specifically relies on temporal reuse of DNN layers, its use is limited to workflows in the DNN domain.

The work by Mood et al[41] enables computation reuse in a distributed environment. It caches secure-function evaluation (SFE) in a buffer to be used by multiple clients. In order to improve scalability, the buffer could be distributed among server nodes. This approach may not be generalized for several domains since the granularity of the reused tasks must be rather coarse to achieve good speedups. The work of Goecks et al[49] also enables reuse in distributed environments with bioinformatics applications. The granularity of reuse is at full application, which limits the potential gains with reuse. Another approach for reuse in bioinformatics is presented in the work of Santos and Santos,[50] which employs memoization to store partial results during protein comparisons.

Connors and Hwu[38] exploit value locality with a combination of a hardware buffer and a profile-guided compiler that groups instructions into reusable tasks as to optimize their granularity. This is a flexible and efficient approach in terms of reuse. However, the implementation may be complex because of the extensions required in the hardware, compiler, and profiling that cooperate to provide the solution.

Guo et al[51] propose to reduce the computation cost of image and audio recognition applications through approximate computation reuse. In their work, IoT devices can attain

reduced computation latency and energy consumption by using a distributed cache system that attempts to reuse results of similar functions previously executed. Although it has shown to be efficient, this approach can only be employed for applications that tolerate approximate results for coarser-grained operations.

In the work of Xu et al,[52] an analytic approach has been proposed for the framework of Isogeometric Analysis (IGA) in which complex matrix calculations may be reused. In essence, it reuses three-dimensional fully calculated models with similar semantic features. The template model, which was previously solved, can export some matrix solutions (when calculating the stiffness matrix) that are proven to be reusable for these similar models. Given that this reuse method was developed for IGA applications and it relies on their features, its use on other domains may be limited.

Li et al[53] Liam tackle the problem of optimizing parameter tuning applications through computation reuse. On their work, the applications being tuned are represented as directed acyclic graphs (DAGs), which may have their atomic stages representation reused at runtime. This reuse is conditional to finding the results cached. They integrate the computation reuse and parameter tuning by steering the generation of an execution DAG containing all pipeline executions for parameter values evaluated. The behavior of this approach on large-scale distributed execution environments was not discussed; it only handles coarse-grain tasks with a memoization-based strategy. These aspects are a limitation to the use in our application domain.

Although a large number of works have been developed in different domains, none of them could be directly applied to our problem. First, we target at having an approach that could be reused in parameter studies, so we have built it on top of a distributed memory domain-specific runtime system. Further, the amount of data employed by these application is very high; as such, classic memoization would not be a practical solution because of the large size of the sub-products generated by application stages. In our solution, tasks that use the same input are bundled into the same stage instances; thus all data are directly forwarded to them without the need for large caching. As presented in the experimental results, the multilevel reuse results in significant performance improvements to our use case application, whereas most of the works have focused on either fine-grain–only or coarse-grain–only approaches.

7 CONCLUSIONS

The execution of SA applied to the context of pathology image analysis has been shown to be an effective tool for improving the understanding of applications on this domain.[14] Although this class of applications could strongly benefit from SA, its use in practice is limited due to the high computational demands.

In this paper, we have leveraged high-performance computing platforms equipped with CPUs and accelerators to efficiently execute SA of a complex cancer image analysis application. Our solution includes an efficient parallelization framework with several optimizations. It was empirically shown the actual benefits of using the hybrid configuration of CPU-MIC (using Intel Phi) on a large-scale distributed environment. We have evaluated

multiple strategies for performing computation reuse in SA of pathology image analysis applications, which resulted in strong performance improvements (2.9×) as compared to not reusing computation. Further, we have shown that, although the RTMA can be effectively used on most cases, its performance degrades on certain scaling cases (see Section 5.3), making the new TRTMA the best all-round choice for computation reuse optimization. As such, the combination of all these optimizations led to a very efficient execution platform that enables the systematic execution of SA in our target application domain.

As future work, we plan to develop novel algorithms for task level merging and evaluate the proposed strategies in other application domains. We will also leverage features found in the related work to improve our approaches. For instance, we are interested in investigating profiling-based approaches to automatically define tasks/stages granularities. The target in this case will be to define granularities that maximize performance, considering data transfers and the benefits of reuse with specific application partitions/decompositions.

ACKNOWLEDGMENTS

This work was supported in part by U24CA180924, U24CA215109, and 1UG3CA225021 from the NCI, R01LM011119-01 and R01LM009239 from the NLM, CNPq, Capes/Brazil grant PROCAD-183794, and NIH K25CA181503. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. Specifically, it used the Bridges system, which is supported by NSF award number ACI-1445606, at the Pittsburgh Supercomputing Center (PSC).

REFERENCES

1. Ioannou A, Itard L. Energy performance and comfort in residential buildings: sensitivity for building parameters and occupancy. Energy Build. 2015;92:216–233.
2. Hamby DM. A review of techniques for parameter sensitivity analysis of environmental models. Environ Monit Assess. 1994;32(2):135–154. [PubMed: 24214086]
3. Morris MD. Factorial sampling plans for preliminary computational experiments. Technometrics. 1991;33(2):161–174.
4. Weirs VG, Kamm JR, Swiler LP, et al. Sensitivity analysis techniques applied to a system of hyperbolic conservation laws. Reliab Eng Syst Saf. 2012;107:157–170.
5. Campolongo F, Cariboni J, Saltelli A. An effective screening design for sensitivity analysis of large models. Environ Model Softw. 2007;22(10):1509–1518.
6. Iooss B, Lemaître P. A review on global sensitivity analysis methods In: Dellino G, Meloni C, eds. Uncertainty Management in Simulation-Optimization of Complex Systems: Algorithms and Applications. New York, NY: Springer Science + Business Media; 2015.
7. Kong J, Cooper LAD, Wang F, et al. Machine-based morphologic analysis of glioblastoma using whole-slide pathology images uncovers clinically relevant molecular correlates. PLoS ONE. 2013;8(11):e81049. [PubMed: 24236209]
8. Mezheyeuski A, Hrynchyk I, Karlberg M, et al. Image analysis-derived metrics of histomorphological complexity predicts prognosis and treatment response in stage II-III colon cancer. Scientific Reports. 2016;6:36149. [PubMed: 27805003]
9. Kothari S, Phan JH, Stokes TH, Wang MD. Pathology imaging informatics for quantitative analysis of whole-slide images. J Am Med Inf Assoc JAMIA. 2013;20(6):1099–1108.
10. Irshad H, Veillard A, Roux L, Racoceanu D. Methods for nuclei detection, segmentation, and classification in digital histopathology: a review—current status and future potential. IEEE Rev Biomed Eng. 2014;7:97–114. [PubMed: 24802905]
11. Yuan Y, Failmezger H, Rueda OM, et al. Quantitative image analysis of cellular heterogeneity in breast tumors complements genomic profiling. Sci Transl Med. 2012;4(157):157ra143.

12. Xing F, Yang L. Robust nucleus/cell detection and segmentation in digital pathology and microscopy images: a comprehensive review. IEEE Rev Biomed Eng. 2016;9:234–263. [PubMed: 26742143]
13. Xu Y, Jia Z, Ai Y, et al. Deep convolutional activation features for large scale brain tumor histopathology image classification and segmentation. Paper presented at: 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP); 2015; Brisbane, Australia.
14. Teodoro G, Kurç T, Taveira LFR, et al. Algorithm sensitivity analysis and parameter tuning for tissue image segmentation pipelines. Bioinformatics. 2017;33(7):1064–1072. [PubMed: 28062445]
15. Barreiros W, Teodoro G, Kurç T, Kong J, Melo AC, Saltz J. Parallel and efficient sensitivity analysis of microscopy image segmentation workflows in hybrid systems. Paper presented at: 2017 IEEE International Conference on Cluster Computing (CLUSTER); 2017; Honolulu, HI.
16. Teodoro G, Pan T, Kurç T, et al. Region templates: data representation and management for high-throughput image analysis. Parallel Computing. 2014;40(10):589–610. [PubMed: 26139953]
17. Saltz J, Gupta R, Hou L, et al. Spatial organization and molecular correlation of tumor-infiltrating lymphocytes using deep learning on pathology images. Cell Reports. 2018;23(1):181–193. e7. [PubMed: 29617659]
18. 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. Paper presented at: 2014 IEEE 28th International Parallel and Distributed Processing Symposium; 2014; Phoenix, AZ.
19. Teodoro G, Pan T, Kurç TM, Kong J, Cooper LAD, Saltz JH. Efficient irregular wavefront propagation algorithms on hybrid CPU-GPU machines. Parallel Computing. 2013;39(4–5):189–211. [PubMed: 23908562]
20. Bradski G The OpenCV Library. 2000 http://www.drdobbs.com/open-source/the-opencv-library/184404319
21. Vincent L Morphological grayscale reconstruction in image analysis: applications and efficient algorithms. IEEE Trans Image Process. 1993;2:176–201. [PubMed: 18296207]
22. Oliveira VMA, Alencar Lotufo R. A study on connected components labeling algorithms using GPUs. Paper presented at: 23rd SIBGRAPI - Conference on Graphics, Patterns and Images; 2010; Gramado, Brazil.
23. Ruifrok AC, Johnston DA. Quantification of histochemical staining by color deconvolution. Anal Quant Cytol Histol. 2001;23(4):291–299. [PubMed: 11531144]
24. Teodoro G, Kurç 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. Int J High Perform Comput Appl. 2017;31(1):32–51. [PubMed: 28239253]
25. Bosilca G, Bouteiller A, Herault T, et al. Performance portability of a GPU enabled factorization with the DAGuE framework. Paper presented at: 2011 IEEE International Conference on Cluster Computing; 2011; Austin, TX.
26. Bueno J, Planas J, Duran A, et al. Productive programming of GPU clusters with OmpSs. Paper presented at: 2012 IEEE 26th International Parallel and Distributed Processing Symposium; 2012; Shanghai, China.
27. He B, Fang W, Luo Q, Govindaraju NK, Wang T. Mars: a MapReduce framework on graphics processors. Paper presented at: 2008 International Conference on Parallel Architectures and Compilation Techniques (PACT); 2008; Toronto, Canada.
28. Linderman MD, Collins JD, Wang H, Meng TH. Merge: a programming model for heterogeneous multi-core systems. SIGPLAN Notices. 2008;43(3):287–296.
29. Mittal S, Vetter JS. A survey of CPU-GPU heterogeneous computing techniques. ACM Comput Surv. 2015;47(4):69.
30. Aji A, Teodoro G, Wang F. Haggis: Turbocharge a MapReduce based spatial data warehousing system with GPU engine. In: Proceedings of the 3rd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data; 2014; Dallas, TX.

31. Saltelli A, Tarantola S, Campolongo F, Ratto M. Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models. Chichester, UK: Wiley; 2004.
32. McKay MD, Beckman RJ, Conover WJ. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. Technometrics. 1979;42(1):55–61. ISBN 00401706.
33. Wolstencroft K, Haines R, Fellows D, et al. The Taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. Nucleic Acids Res. 2013;41(W1):W557–W561. [PubMed: 23640334]
34. Stoer M, Wagner F. A simple min-cut algorithm. J ACM. 1997;44(4):585–591.
35. Goldschmidt O, Hochbaum DS. Polynomial algorithm for the k-cut problem. In: [Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science; 1988; White Plains, NY.
36. Halton JH. Algorithm 247: radical-inverse quasi-random point sequence. Commun ACM. 1964;7(12):701–702.
37. Sobol IM. Uniformly distributed sequences with an additional uniform property. USSR Comput Math Math Phys. 1976;16(5):236–242.
38. Connors DA, Hwu W-MW. Compiler-directed dynamic computation reuse: Rationale and initial results. In: Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture; 1999; Haifa, Israel.
39. Lepak KM, Lipasti MH. On the Value locality of store instructions. SIGARCH Comput Archit News. 2000;28(2):182–191.
40. Wang W, Raghunathan A, Jha NK. Profiling driven computation reuse: An embedded software synthesis technique for energy and performance optimization. In: Proceedings of the 17th International Conference on VLSI Design; 2004; Mumbai, India.
41. Mood B, Gupta D, Butler K, Feigenbaum J. Reuse it or lose it: More efficient secure computation through reuse of encrypted values. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security; 2014; Scottsdale, AZ.
42. Steen JV, Coenders JL, Pasterkamp S, Rolvink A, Steekelenburg JV. Computational reuse optimisation for stadium design. In: Proceedings of the International Association for Shell and Spatial Structures; 2015; Amsterdam, The Netherlands.
43. Nakra T, Gupta R, Soffa M. Value prediction in VLIW machines. In: Proceedings of the 26th Annual International Symposium on Computer Architecture; 1999; Atlanta, GA.
44. Sodani A, Sohi GS. Dynamic instruction reuse. In: Proceedings of the 24th Annual International Symposium on Computer Architecture; 1998; Denver, CO.
45. Richardson SE. Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation. Technical Report. Mountain View, CA: Sun Microsystems, Inc; 1992
46. Alvarez C, Corbal J, Valero M. Fuzzy memoization for floating-point multimedia applications. IEEE Trans Comput. 2005;54(7):922–927.
47. Modarressi M, Nikounia SH, Jahangir A-H. Low-power arithmetic unit for DSP applications. Paper presented at: 2011 International Symposium on System on Chip (SoC); 2011; Tampere, Finland.
48. Riera M, Arnau JM, González A. Computation Reuse in DNNs by Exploiting Input Similarity ISCA ‘18. Piscataway, NJ: IEEE Press; 2018:57–68.
49. Goecks J, Nekrutenko A, Taylor J. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biology. 2010;11(8):R86. [PubMed: 20738864]
50. Santos EE, Santos E Jr. Effective computational reuse for energy evaluations in protein folding. Int J Artif Intell Tools. 2006;15(5):725–739.
51. Guo P, Hu B, Li R, Hu W. FoggyCache: Cross-Device Approximate Computation Reuse MobiCom ‘18. New York, NY: ACM; 2018:19–34.
52. Xu G, Kwok T-H, Wang CCL. Isogeometric computation reuse method for complex objects with topology-consistent volumetric parameterization. Comput-Aided Des. 2017;91:1–13.
53. Li L, Sparks ER, Jamieson KG, Talwalkar A. Exploiting reuse in pipeline-aware hyperparameter tuning. 2019 arXiv preprint arXiv:1903.05176.

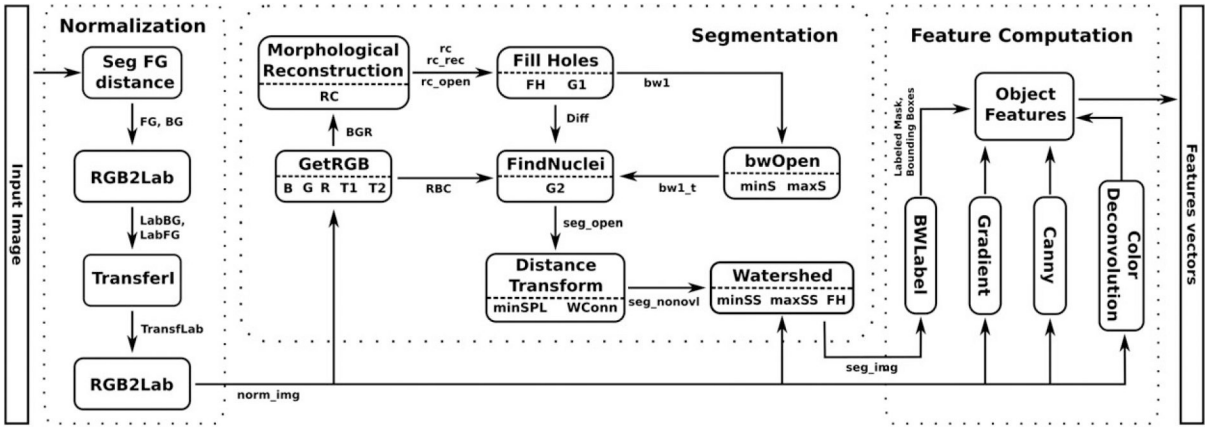

FIGURE 1.

The workflow of the motivating microscopy image analysis application with the high-level stages (Normalization, Segmentation, and Feature Extraction) and their internal operations. The input parameters used by the operations in the Segmentation stage are also shown in boxes below the name of the operations. The inter-task data transfers are presented close to the arrows pointing the origin and destination of the data

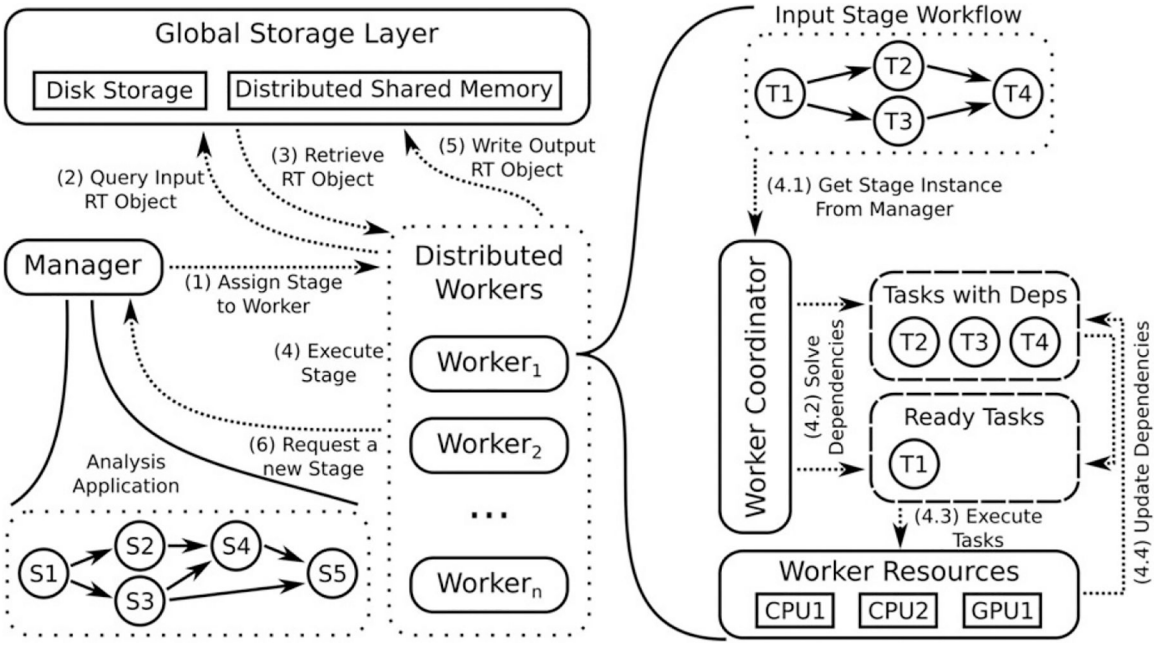

FIGURE 2.

Overview of the Region Templates framework architecture and execution scheme. On the left side of the Figure, we have a global view of the framework (distribution of stages from the manager to the worker nodes), whereas on the right side, the execution on the worker node side is presented. The process of executing a single stage, from its assignment to the request of another stage, is further detailed[16]

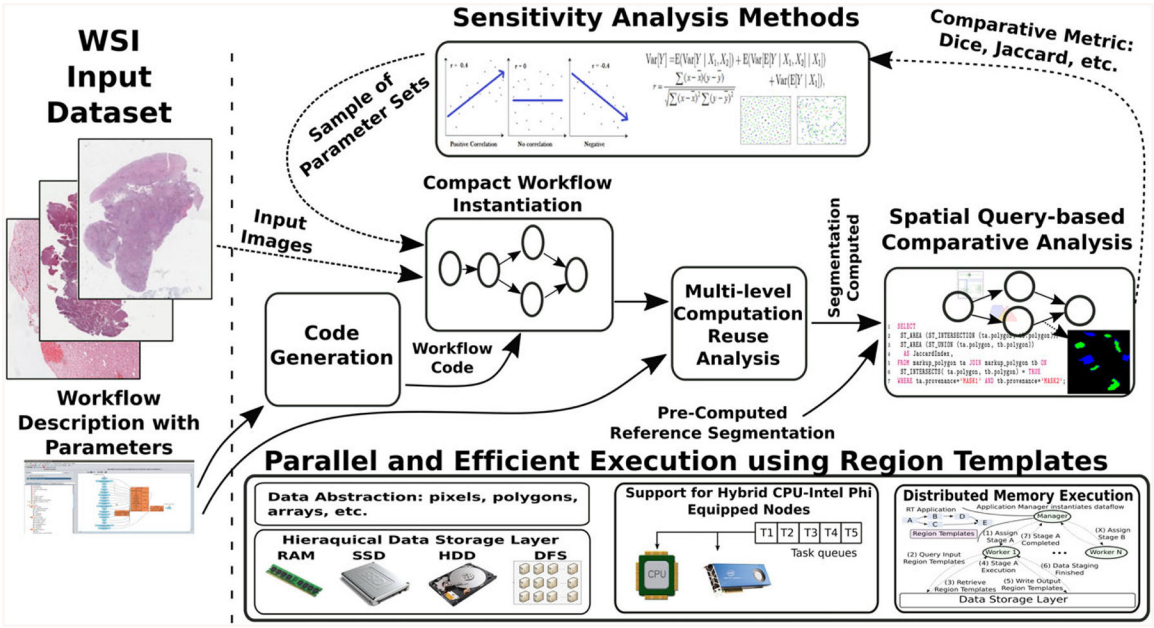

FIGURE 3.

The parameter SA framework. The system receives an application and parameters to be studied along with input data and the selected SA method as input. It then generates the application code and executes it on top of the Region Templates system using parallel machines. Results from segmentation outputs for different parameter values are compared using an optimized spatial indexing, and these differences or detected variations in the output results are used by the SA method to compute sensitivity of the output with respect to input parameter changes

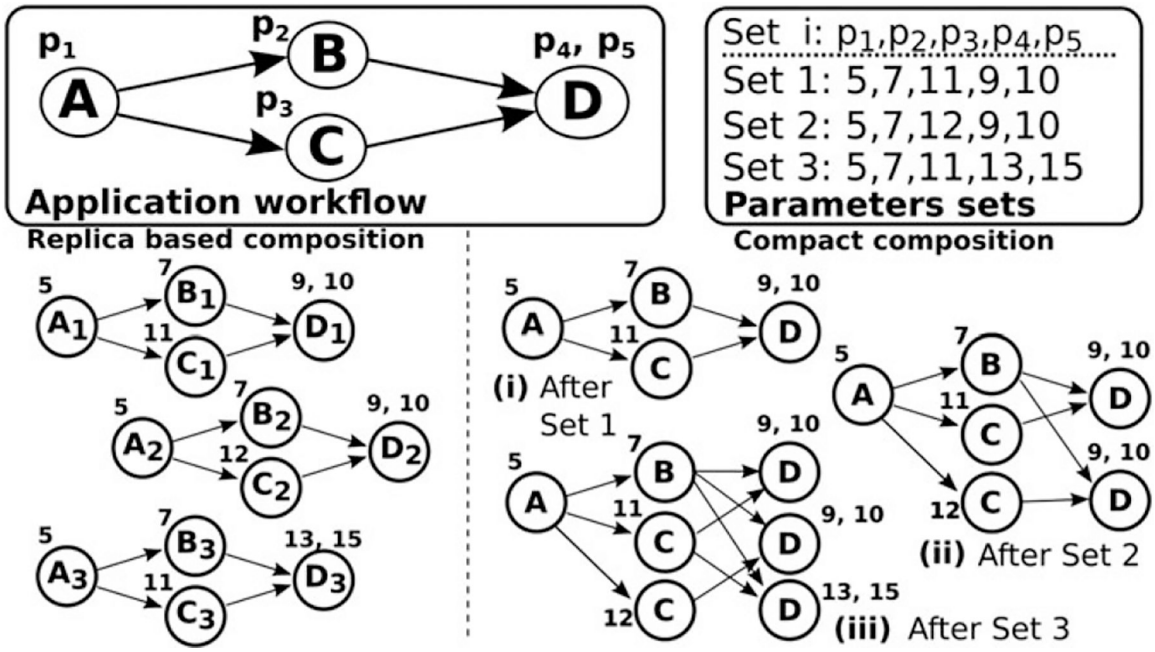

FIGURE 4.

Example of a workflow composition based on a simple application. The application is described as a workflow of stages, each stage having as its inputs their respective parameters (p_1-p_5) and the output data of the previous stages (if there is any). This application is instantiated with three parameter sets, which can be composed either by fully replicating the application workflow for each parameter set, or by performing a compact composition, which avoids replicating stages that would return the same result (ie, same input data and parameters)

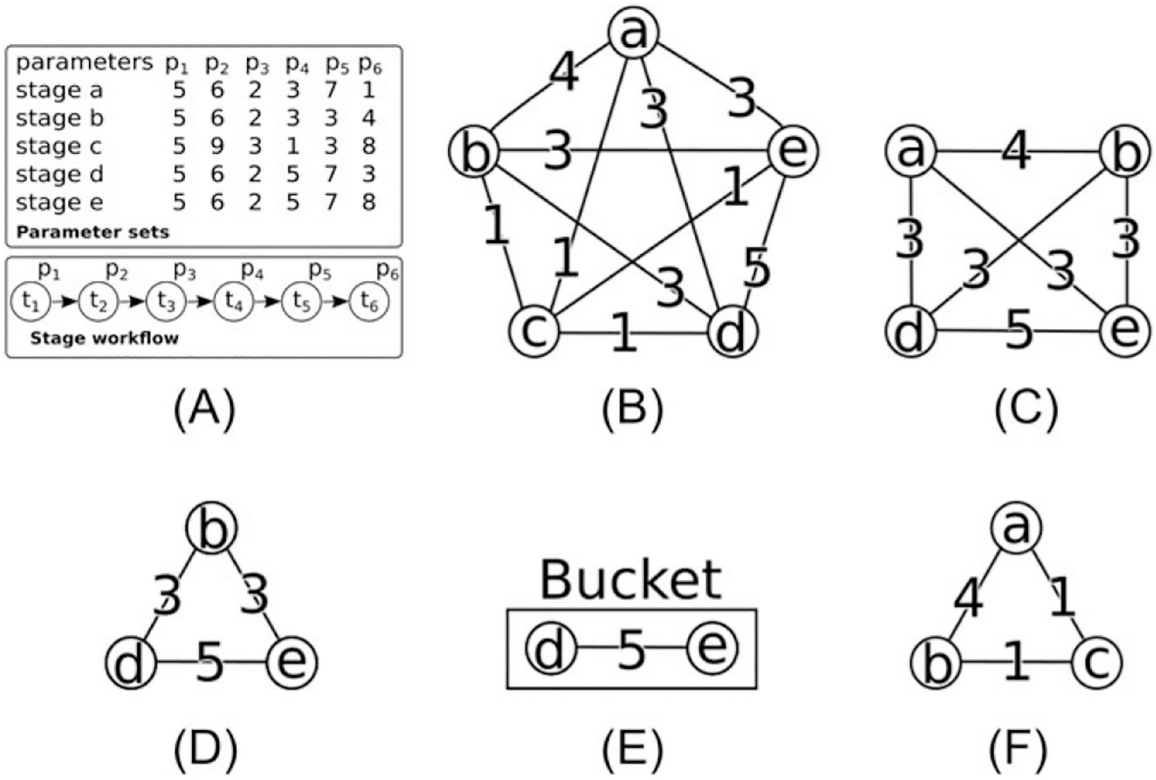

FIGURE 5.

An example on which SCA executes on five instances of a workflow application of six tasks, with *MaxBucketSize* = 2. On this depiction of the merging problem, each node represents a stage instance (same stages with different parameters) with the edges as the degree of reuse between two stage instances. A, Example application with the stages' parameters instantiations; B, Initial graph of the example with all stages instances and their respective degree of reuse; C, First cut is performed by removing node c, which is a cut of weight 4; D, After the next min-cut of weight 10, node a is removed; E, After final cut of node b, a MaxBucketSize sized subgraph is found and set apart on the solution list; F, After the formation of a bucket the cutting process starts over with the remaining nodes. This procedure continues until all nodes are assigned to a bucket

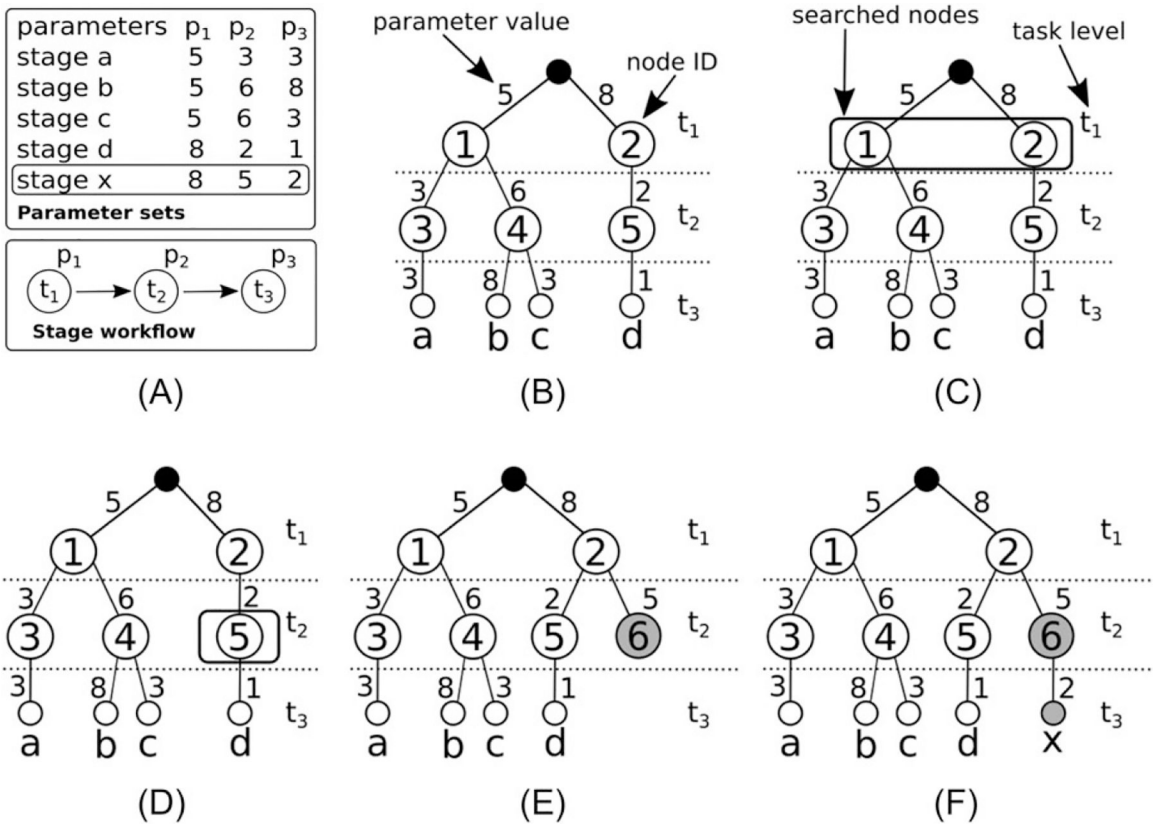

FIGURE 6.

An example in which node *x* is inserted on the existing reuse-tree. Figure 6A defines the tasks of which each stage is composed by and presents the parameters' values for each stage instance. A, Example application; B, Initial reuse-tree for the instance example; C, Searching for reuse on the first task; D, Searching for reuse on the second task; E, Inserting a new node 6; F, Inserting the leaf node x

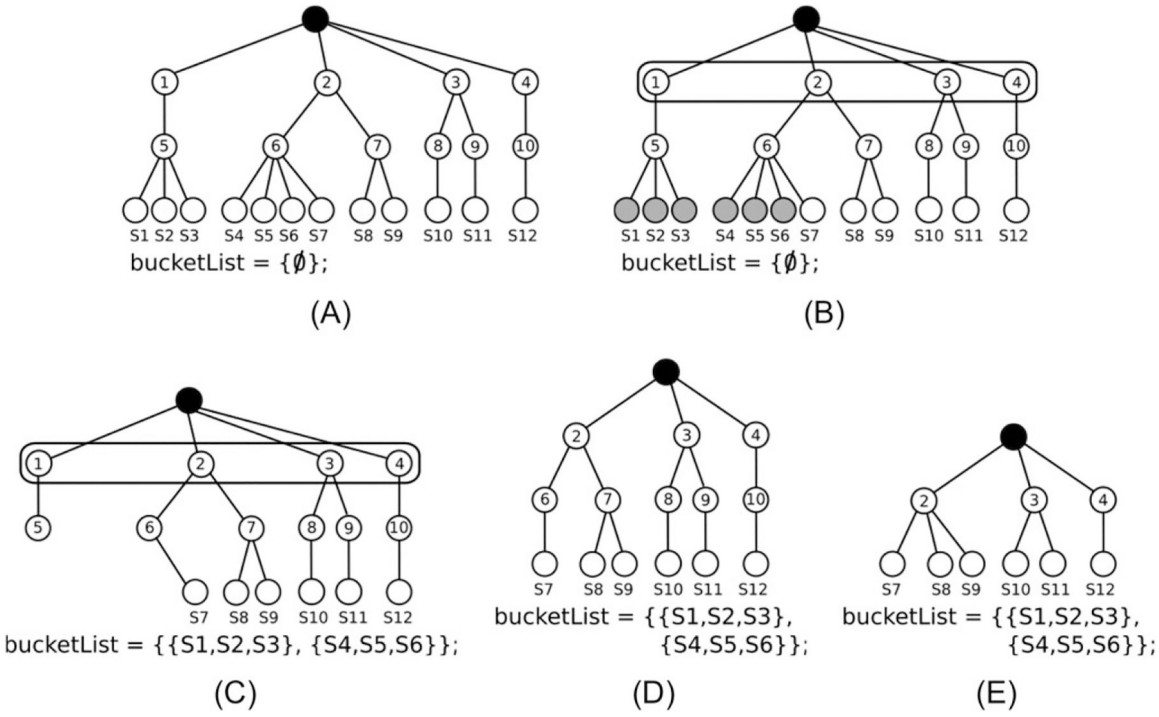

FIGURE 7.

An example of RTMA with MaxBucketSize 3. The merged stages of each step are shown below the tree on the bucket list. A, Initial reuse-tree; B, Reuse-Tree after select procedure; C, Reuse-Tree after the selected merged leaf nodes are pruned and added to the bucket list; D, Reuse tree after the childless parents are recursively removed; E, Reuse tree after move-up procedure

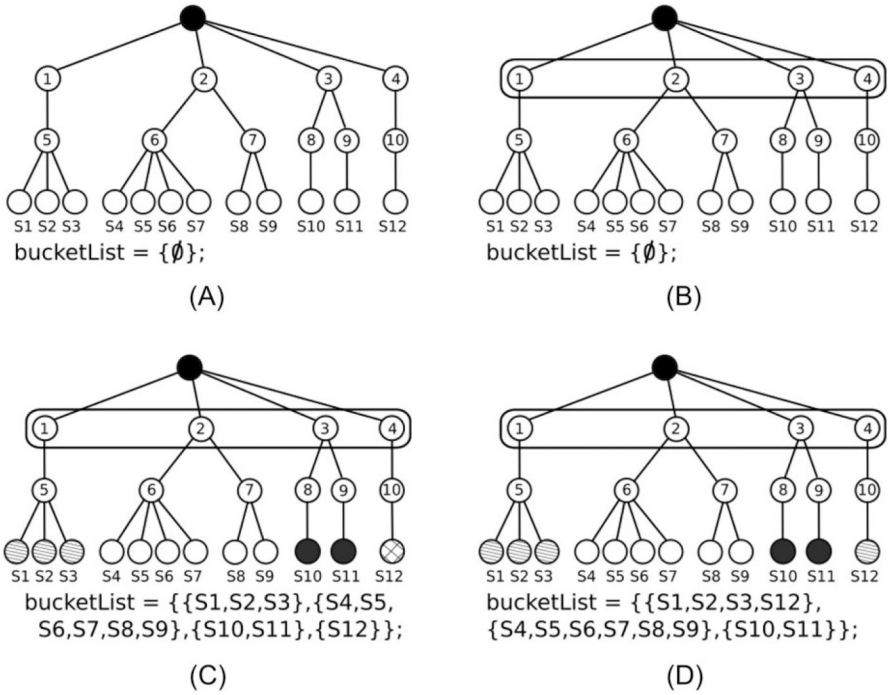

FIGURE 8.

An example of the buckets generated by the first two phases of the algorithm: Full-Merge and Fold-Merge, using *MaxBuckets* = 3. A, Initial reuse-tree; B, Attempt of Full-Merge from root node results in four buckets, which is greater than MaxBuckets = 3; C, After Full-Merge, the minimum number of four buckets are created. Still, we need to be reduce the number of buckets to MaxBuckets through Fold-Merging; D, Fold-Merge, the merges first and last buckets of the previous tree to achieve the correct number of three buckets

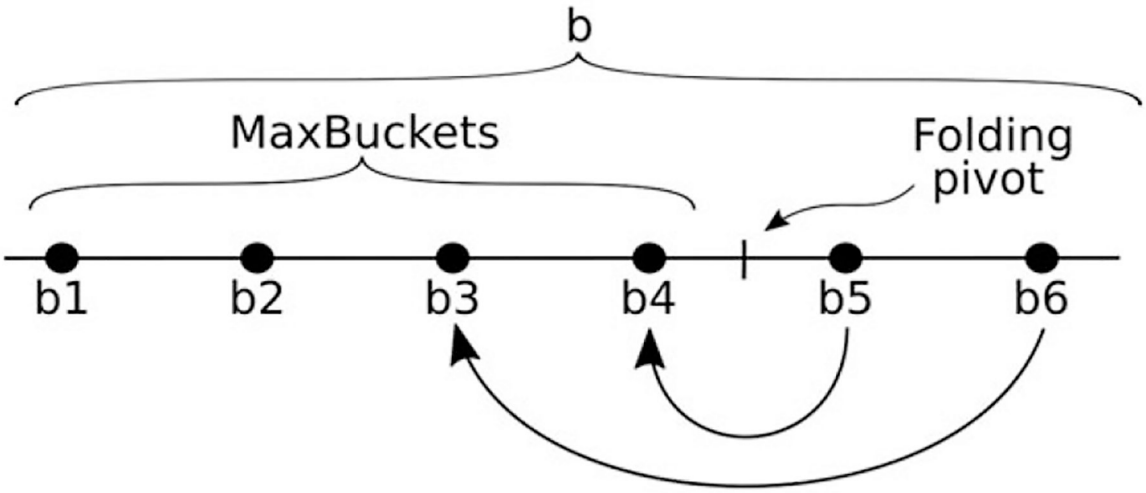

FIGURE 9.

An example of a Fold-Merging of buckets b1–b6. Initially, we start with $b = 6$ buckets, trying to achieve $MaxBuckets = 4$ buckets. The task cost of the buckets follows the ordering $b1 \geq b2 \geq b3 \geq b4 \geq b5 \geq b6$. The pairs of buckets b4 and b5, as well as b6 and b3, are merged together

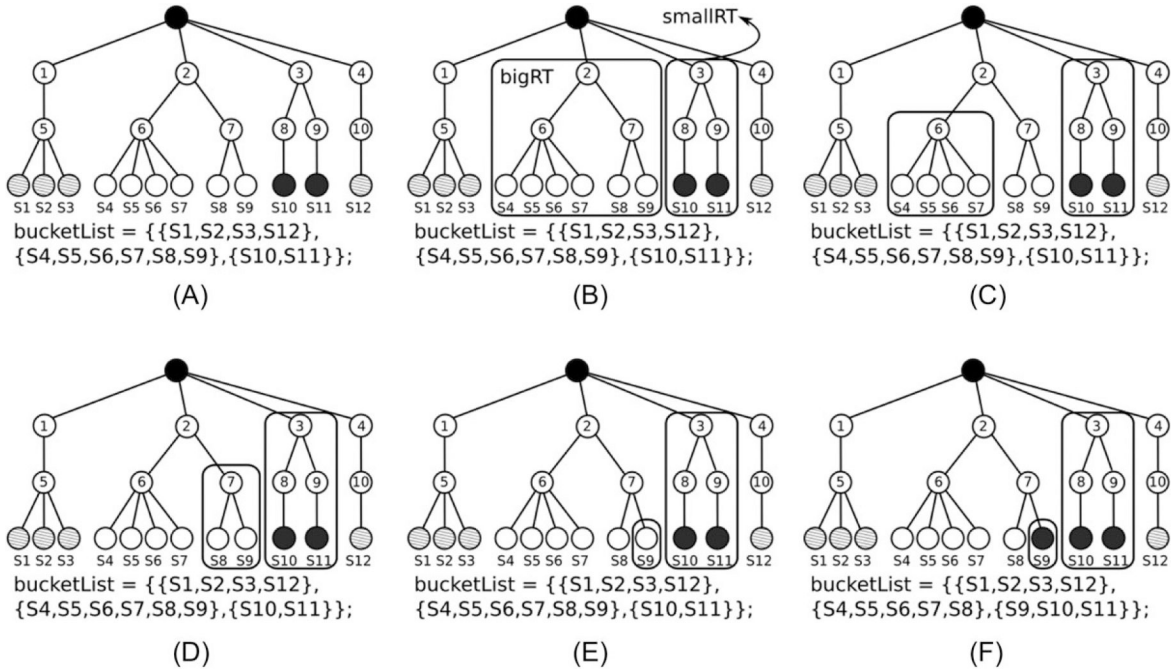

FIGURE 10.

An example of the Balance step on which there are 3 buckets to be balanced. A, Initial reuse-tree with 3 buckets of costs 8, 9, and 5, respectively; B, Buckets of greatest and smallest costs are selected, with current imbalance of 4 and max cost 9; C, The algorithm attempts to send node 6 to smallRT, but it is aborted because the result has a greater imbalance of 7; D, The attempt with node 7 is also done, and this is also aborted because a bucket with cost 9 (not smaller than the original solution) is created; E, By sending node S9 to smallRT, we have an imbalance of 0 and max cost 8, making it a viable balancing operation; F, After the balancing operation of sending node S9 to smallRT, we have the buckets with updated costs 8, 8, and 8

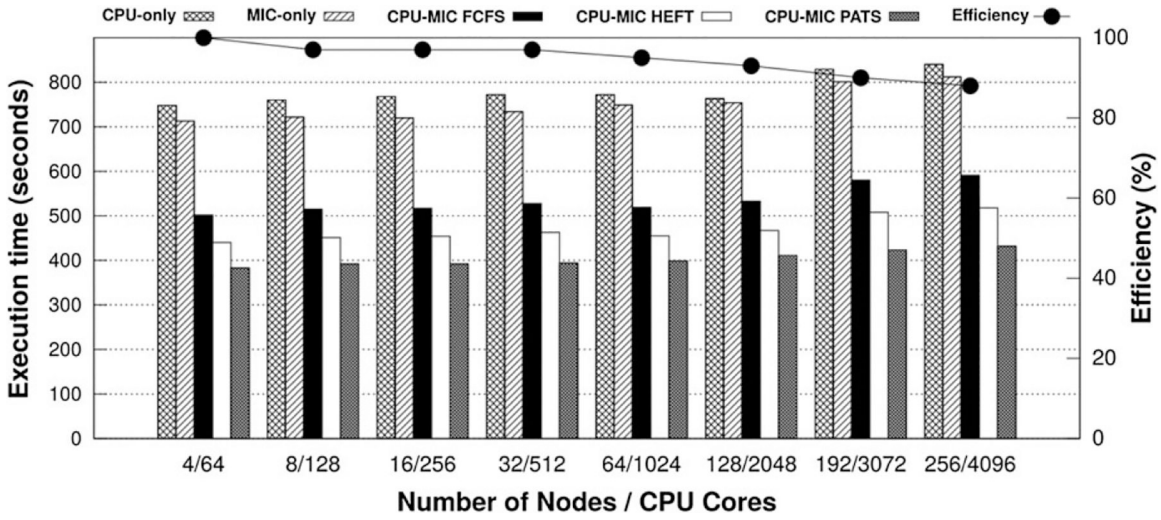

FIGURE 11.

Performance of the application in a weak scaling execution for multiple application versions and scheduling strategies. The number of MICs is the same as the Number of Nodes, with all 16 CPU cores of each Stampede node being used when this processor is employed

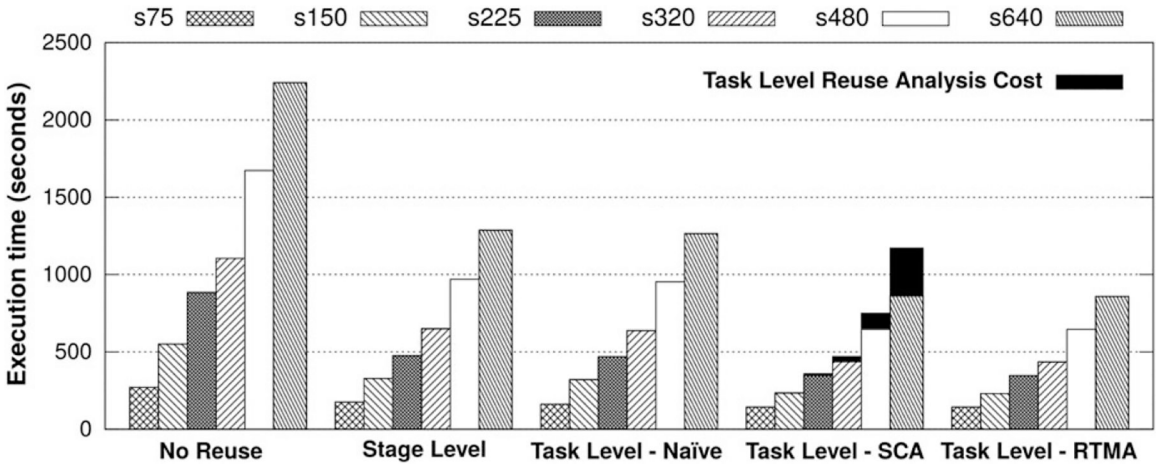

FIGURE 12.

Performance of multiple computation reuse algorithms for a MOAT SA experiment as the parameter set sample size is varied from 75 (s75) to 640 (s640)

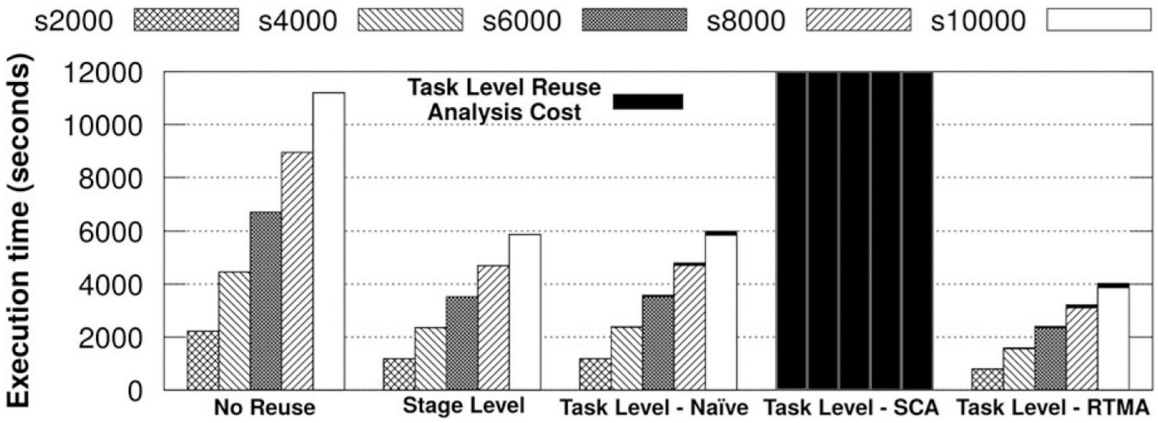

FIGURE 13.

Impact of the computation reuse strategies for the VBD SA method as parameter set sample size is varied from 2000 (s2000) to 10 000 (s10000)

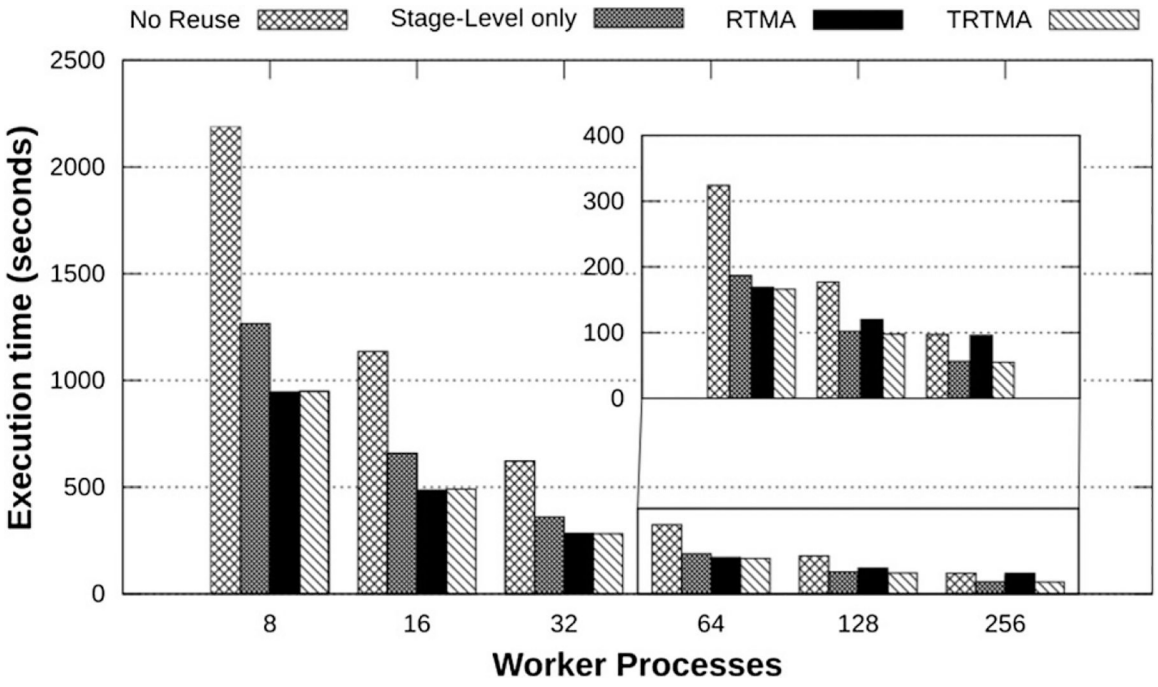

FIGURE 14.

Performance of different reuse strategies for MOAT with 1000 parameter samples. The scalability of RTMA is compromised due to the merging

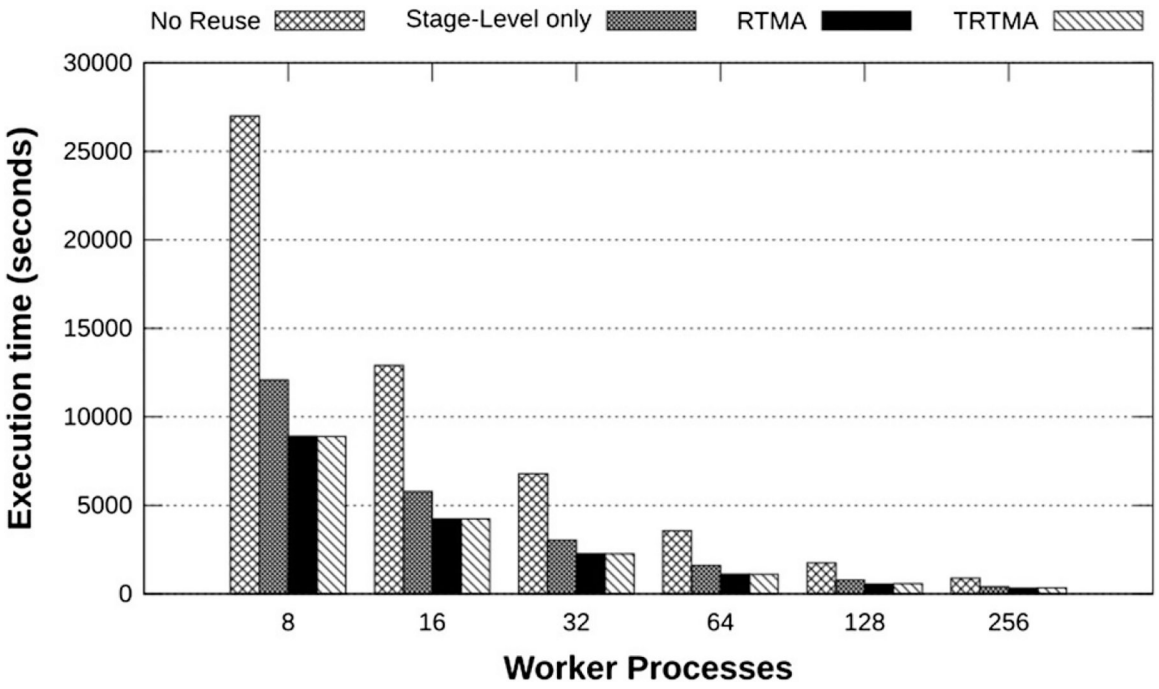

FIGURE 15.
Performance of different reuse strategies for VBD with 10 000 parameter samples

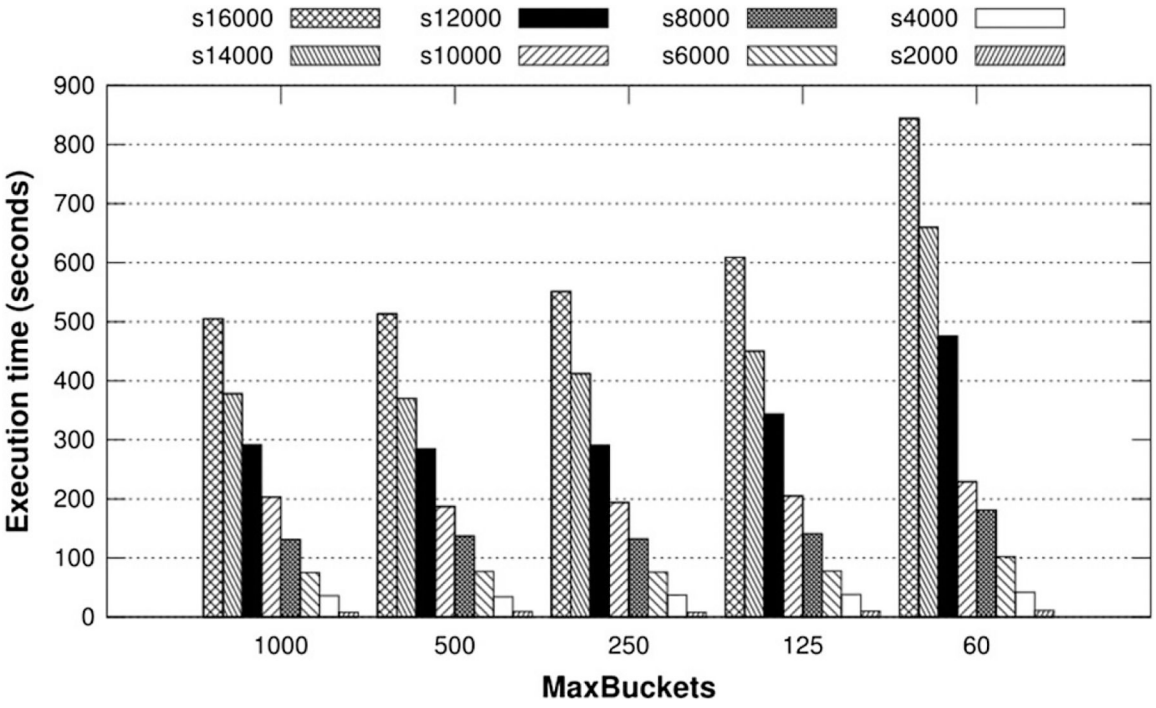

FIGURE 16.

Execution time of TRTMA for different as the parameter sample size and *MaxBuckets* are varied

TABLE 1

List of parameters used by operations in the Segmentation stage of our motivating application and their range values. The search space formed is of 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	Thresholds to identify candidate nuclei	[5, 10, …, 80]
G2	Thresholds to identify candidate nuclei	[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]

TABLE 2

Operations used in each of the application main stages and their parallelism strategy

Operations	Description	Source code ref.
	Normalization phase	
Seg FG dist	Segment foreground from background w/ discriminant functions	Ours
RGB2LAB	Convert from RGB to LAB	Ours
TransferI	Map color distribution of an image to that of the target image	Ours
LAB2RGB	Convert from LAB to RGB	Ours
	Segmentation phase	
GetRBC	Covert RGB image to grayscale and estimate background coverage	OpenCV[20]
Morphological open	Opening removes small objects and fills holes in foreground	OpenCV[20]
Morphological reconstruction	Flood-fill a marker image that is limited by a mask image	Vincent[21]
Area threshold	Remove objects outside an area range	IWPP[19]
Fill holes	Fill holes objects w/ a flood-fill starting at selected points	Vincent[21]
Distance transform	Compute min distance from foreground pixels to background	Ours
Watershed	Separate overlapping objects	OpenCV[20]
	Feature extraction phase	
BWLabel	Label components (objects) of a mask image with the same value	Oliveira[22]
Color deconvolution[23]	Separate multistained biological images in different channels	Ours
Gradient	Compute image gradient in x,y	OpenCV[20]
Sobel edge	Compute Sobel edge	OpenCV[20]
Object features	Compute statistics (mean, median, max, etc) for each object	Ours

TABLE 3

Example output of a MOAT analysis with all 15 parameters and a VBD analysis with a selection of the eight most influential parameters. The influence of a parameter is bounded in the interval [−1,1] and is proportional to its distance from 0 (ie, 1 and −1 are the greatest values and 0 the smallest)

Parameter	MOAT Effect	VBD	
		First-order effect (main)	Higher-order effects (total)
B	−0.0108	-	-
G	−0.0064	-	-
R	−0.0189	-	-
T1	0.0207	-	-
T2	0.0417	0.0006	0.0001
G1	0.8157	0.2251	0.2371
G2	0.9197	0.7305	0.7886
MinSize	0.0889	0.0025	0.0056
MaxSize	0.1820	0.0150	0.0086
MinSizePI	0.0341	0.0021	0.0022
MinSizeSeg	−0.0155	-	-
MaxSizeSeg	−0.0184	-	-
FillHoles	−0.0276	-	-
MorphRecon	0.1321	0.0146	0.0149
Watershed	0.0530	0.0018	0.0016

TABLE 4

Specification of the images and the native database used as test inputs

Images source:	**The Cancer Genome Atlas (TCGA) database**
Images type:	Glioblastoma Multiforme (GBM) whole slide tissue image (WSI)
Images size:	Approximately 120K×120K pixels
Images tiling:	About 136,568 tiles of 4K×4K pixels

TABLE 5

Maximum computation reuse potential for the parameter sampling methods MC, LHS, and QMC with different parameter sample sizes. The reuse percentages represent reuse only at fine-grain reuse after coarse-grain reuse is computed

Sampling strategy	Number of nodes		
	2000	6000	10000
MC	36.35%	36.46%	36.40%
LHS	36.62%	36.44%	36.44%
QMC	35.10%	34.44%	33.48%

TABLE 6

Comparative of computation reuse approaches

Reference	Granularity	Reuse strategy	Hardware independent	Large Scale dataset	Distributed
[44]	Fine-grain	Memoization	✗	✗	✗
[45]	Fine-grain	Memoization	✗	✗	✗
[46]	Fine-grain	Memoization	✗	✗	✗
[47]	Fine-grain	Memoization	✗	✗	✗
[48]	Fine-grain	Analytic	✗	✓	✗
[40]	Fine-grain	Analytic Memoization	✗	✗	✗
[41]	Coarse-grain	Memoization	✓	✗	✓
[49]	Coarse-grain	Memoization	✓	✗	✓
[50]	Coarse-grain	Memoization	✓	✗	✗
[38]	Coarse-grain Fine-grain	Memoization	✗	✗	✗
[51]	Coarse-grain	Memoization	✓	✗	✓
[52]	Coarse-grain	Analytic	✓	✗	✗
[53]	Coarse-grain	Memoization	✓	✓	✗
Our Work	Coarse-grain Fine-grain	Analytic	✓	✓	✓