Published in final edited form as:

Neuroinformatics. 2014 October ; 12(4): 563–573. doi:10.1007/s12021-014-9234-5.

Resource Estimation in High Performance Medical Image Computing

Rueben Banalagay,
Electrical Engineering, Vanderbilt University EECS, 2301 Vandervilt P1, PO Box 351679 Station B, Nashville 37235-1679, TN, USA

Kelsie Jade Covington,
Electrical Engineering, Vanderbilt University EECS, 2301 Vandervilt P1, PO Box 351679 Station B, Nashville 37235-1679, TN, USA

D.M. Wilkes, and
Electrical Engineering, Vanderbilt University EECS, 2301 Vandervilt P1, PO Box 351679 Station B, Nashville 37235-1679, TN, USA

Bennett A. Landman
Electrical Engineering, Vanderbilt University EECS, 2301 Vandervilt P1, PO Box 351679 Station B, Nashville 37235-1679, TN, USA. Biomedical Engineering, Vanderbilt University, Nashville 37235, TN, USA

Bennett A. Landman: bennett.landman@vanderbilt.edu

Abstract

Medical imaging analysis processes often involve the concatenation of many steps (e.g., multi-stage scripts) to integrate and realize advancements from image acquisition, image processing, and computational analysis. With the dramatic increase in data size for medical imaging studies (e.g., improved resolution, higher throughput acquisition, shared databases), interesting study designs are becoming intractable or impractical on individual workstations and servers. Modern pipeline environments provide control structures to distribute computational load in high performance computing (HPC) environments. However, high performance computing environments are often shared resources, and scheduling computation across these resources necessitates higher level modeling of resource utilization. Submission of 'jobs' requires an estimate of the CPU runtime and memory usage. The resource requirements for medical image processing algorithms are difficult to predict since the requirements can vary greatly between different machines, different execution instances, and different data inputs. Poor resource estimates can lead to wasted resources in high performance environments due to incomplete executions and extended queue wait times. Hence, resource estimation is becoming a major hurdle for medical image processing

Correspondence to: Bennett A. Landman, bennett.landman@vanderbilt.edu.

Information Sharing Statement

This work is available in open source and within the JIST platform (RRID:nlx_151344) through the Neuroimaging Informatics Tools and Resources Clearinghouse (NITRC) project "JIST." All source code is at https://www.nitrc.org/projects/jist/. The Hypervisor is stored in the "Hyperadvisor" module in the source code repository. A public instance of the Hypervisor is available via the default settings of JIST.

algorithms to efficiently leverage high performance computing environments. Herein, we present our implementation of a resource estimation system to overcome these difficulties and ultimately provide users with the ability to more efficiently utilize high performance computing resources.

Keywords

Java image science toolkit; JIST RRID:nlx_151344; Resource estimation; High performance computing; Decision trees

Introduction

With the increased use of more complex algorithms on progressively larger amounts of medical image data, there is a growing need to use high performance computing environments, such as grid clusters, to process results quickly and efficiently (e.g., (Rex et al. 2003; Pieper et al. 2006; Sheehan et al. 1996; Parker and Johnson 1995; Lucas et al. 1992; Konstantinides and Rasure 1994; Lucas et al. 2010)). However, high performance computing (HPC) environments are often shared resources, and submitting software tasks requires an estimate of the CPU runtime and memory usage needed for execution. Moreover, higher level data management systems (e.g., XNAT database, pipeline, and web portal (Marcus et al. 2005)) are increasingly being used to manage both archival and pipeline process; the full automation and potentially heterogeneous processing tasks further complicate accurate resource estimation.

Neuroscience pipelines (LONI Pipeline (Rex et al. 2003), JIST (Lucas et al. 2010), NyPipe, NA-MIC Tools (Pieper et al. 2006), AVS (Sheehan et al. 1996), SCIRun (Parker and Johnson 1995), Koros (Konstantinides and Rasure 1994), NiPype (Gorgolewski et al. 2011), XNAT pipelines (Marcus et al. 2005), etc.) run on top of HPC environments (e.g. Sun Grid Engine, Oracle Redwood City, CA). In single user settings (i.e., a small cluster of nodes within a research lab), these environments may be configured to ignore resource quotas. However, in larger deployments resource scheduling is essential to ensure fair allocation of shared resources among a diverse user base. The grid management software has access to the control files and binaries that are used by each process, but does not have any context with which to interpret the data that will be processed. Therefore, the state of the art is to ask the user to specify the anticipated memory and time usage for their processes. Therefore, the requirement for resource estimation is increasingly imposing a challenge for efficiently leveraging high performance computing. Accurate estimates are essential as an underestimation results in the premature termination of the submitted task by the high performance control engine, while an overestimation results in longer queue wait times for the task. Both cases would, inevitably, result in resources wasted on either insufficient or unneeded execution.

In contrast to algorithmic worst-case analysis, high performance computing resource estimation requires one to consider more factors than the behavior of the asymptotic behavior of the underlying algorithm. A program’s real-world runtime and memory requirements are also dependent on the current inputs, hardware environment, storage access, and, potentially, external factors (e.g., network conditions). Such dependencies

create difficulties on multiple levels. First, different programs require different inputs, thereby limiting any comparisons between different tasks. In addition, the values of the inputs for an individual task can vary between executions, thus changing the resource requirements needed for completion. Lastly, hardware environments can vary from user to user; while the underlying code stays the same, the different hardware characteristics can lead to different resource requirements on different machines. Thus, such heterogeneities on both the software and hardware level make it difficult to provide a consistent framework in which to estimate the resource requirements of a particular task. Figure 1 illustrates this point, as a single algorithm in a complex multi-stage imaging pipeline has, in itself, varying resource requirements from run to run and from machine to machine.

While software resource estimation is a fairly new topic in the medical image computing community, similar work has been undertaken in estimating CPU resource usage for general workloads submitted to high performance computing environments. Methods used for this problem have included using K-nearest neighbor smoothing (Iverson et al. 1999) and time-series smoothing (Sonmez et al. 2009) of the historical data. Genetic algorithms (Smith et al. 1997) have been used to combine similar runtime instances of a program in order to create local models for that group.

Herein, we advocate that the application specific control logic (the “neuroscience pipelines”) have sufficient access to the complexity such that they can form reasonable estimates of resource usage by monitoring past performance. We implement the proposed approach within the context of JIST, but the estimation framework is designed as a standalone service and does not depend on the particulars of the neuroscience pipeline. We extend and evaluate a preliminary implementation of a “Hypervisor” system for resource estimation in medical imaging. We use the term “Hypervisor” to indicate that the process is supervising/monitoring entity that does not directly manage computation flow (i.e., the “pipeline environment”). Simply put, the Hypervisor assists in resource allocation but does not schedule resources or control logical program flow.

In the software community, a focus of research is to estimate the development effort of new software projects based on the effort outcomes of previous development projects. Among the many machine learning methods used for the problem, the more common approaches include ordinary least squares regression, robust regression, stepwise ANOVA, classification and regression trees, analogy based estimation, case based reasoning, fuzzy systems, artificial neural networks, and support vector machines (Dejaeger et al. 2012; Briand and Wieczorek 2001; Gray and MacDonell 1997). Each of these methods has advantages and disadvantages depending on the context and the data being used to train an estimation model; there is no “best” method for approaching this problem (Briand and Wieczorek 2001). In fact, even the problem of determining if a program will terminate (the halting problem) is provably unsolvable (Turing 2004). Here, we first example the efficacy of baseline prediction methods and focus on using regression/decision trees; a tree structure allows us to meaningfully cluster similar task execution instances without regards to the categorical or ordinal nature of the input variables.

The proposed Hypervisor is evaluated in the context of the Java Image Science Toolkit (JIST) (Lucas et al. 2010). JIST is a cross-platform, open source plugin to MIPAV (Medical Image Processing and Visualization) (McAuliffe et al. 2001), an image processing software from the National Institute of Health. The JIST platform is capable of providing end users with a graphical environment conducive to rapid algorithm development and large scale image processing (Lucas et al. 2010). In particular, we focus on integrating with the Layout Tool and Process Manager functionality within JIST (Fig. 2). The Layout Tool serves as a means to easily drag and drop JIST modules and customize their input and output parameters. Resulting JIST Layouts can then be loaded into the Process Manager for real-time management, execution, and feedback of JIST jobs and batch processes. The goal of our Hypervisor implementation is to leverage the modular nature of JIST layouts; we collect input and resource usage data on individual modules along with the current hardware environment order to create better resource estimates for future module executions. Note that the Hypervisor is implemented separately from the JIST framework, supports ubiquitous Apache Axis 2 business logic, and could be used as a resource manager for other programs with Axis 2 support.

The presented work builds on an earlier version of a Hypervisor that used a K-nearest neighbors approach to cluster similar instances of tasks to create a Gaussian resource estimation model (Covington 2011). The previous version's K-nearest neighbors algorithm requires the creation of distance metrics for comparison, and the practical, varied combinations of scalars, vectors, and strings as part of a task's input resulted in difficulties in creating consistent distance metrics to capture meaningful similarities between the different task execution contexts.

Methods

The Hypervisor system is based on a client–server model between remote JIST users and an Apache Axis 2 webserver (http://axis.apache.org/axis2/java/core/) with a MySQL database (http://www.mysql.com/). The Axis 2 webserver facilitates web communications between the server and JIST clients by allowing both sides to communicate as Plain Old Java Objects (POJO). Note that Axis 2 is a scalable communication protocol that can be accessed without deep knowledge of underlying network technologies. The MySQL database is configured to store the relevant data of previous Hypervisor requests for use in creating resource estimates. Figure 3 gives an overview of the system components and the flow of data within the system.

The Hypervisor server runs on a 64 bit (Ubuntu, Canonical Ltd., Isle of Man) virtual machine running (VirtualBox, Oracle, Redwood Shores, CA) an Apache Axis 2 webserver. Incoming data collected from completed JIST processes are stored in a MySQL database as follows:

- Tables in the database correspond to a unique module.
- Rows in a particular table represent the data from different executions of that module

- Columns of the table represent the inputs of the module along with the machine environment and actual resource requirements of a particular module execution.

The data for each table is used to train regression tree models for the algorithm that each table represents. In effect, the regression tree finds clusters in the table data for use in estimating resource requirements. Due to the relatively slow nature of training regression trees in relation to the need for fast resource estimates upon request, we have split our system into two parts as described below: (1) Offline Training and (2) Online Estimation and Update.

Offline Training

Each table in the database represents one algorithm and is used to train two regression trees — one for estimating the CPU time and one for memory requirements for that algorithm. Figure 4 summarizes the flow of data in the offline training procedure. Each of the recorded variables is considered as a numeric entry in a vector. Some of the entries (also called features) will be predictive of the resource utilization while others will not be. We use a machine learning process to identify salient features and reveal the predictive models. Offline training typically takes less than 10 min on a lightweight virtual machine.

Specifically, the vector $x=[x_1 x_2 \ldots x_m]^T$ contains the values of the recorded inputs to that instance of the algorithm while the y value is a scalar representing the actual recorded CPU time or the used memory needed for execution. These input/output pairs are then stacked into feature vectors $v=[x^T y]^T$ to grow a regression tree in a similar manner as described in Breiman (Breiman et al. 1984). Given a set of feature vectors $T=\{v_1, v_2, \ldots, v_k\}$ at a node N, the algorithm creates m orderings of T with respect to each of the m elements in x. If the i_{th} element of x is ordinal, T is arranged so that all of the x_i elements of T are in ascending order. If x_i is categorical with different categories $c=\{c_1, c_2, \ldots, c_j\}$, we define $\mu(c_a)$ as the average of all y values such that $x_i=c_a$. T is then ordered so that the categories of x_i are in ascending order of $\mu(c_a)$

For each of the resulting m orderings, we find a binary rule of the form $x_i<\beta$ or $x_i \varepsilon S \subset C$ that naturally splits T into a left set T_i and a right set T_r in which the following expression is minimized:

$$\rho_l \times \sigma^2(T_l) + \rho_r \times \sigma^2(T_r) \quad (1)$$

where p_i and p_r are the proportion of the feature vectors in T that are in T_i and T_r, respectively, while $\sigma^2(T_i)$ and $\sigma^2(T_r)$ are the variance of the y values in T_i and T_r, respectively. Note, this criterion performs best when peaks are present in the distribution of y values in T as natural splits can be found by creating clusters around the peaks.

The created m splitting rules are then compared with each other to find the one splitting rule by which Eq. (1) is minimized. That rule is then stored at node N, and child nodes N_i and N_r are added that store the samples in T_i and T_r respectively. The algorithm continues to bifurcate the added nodes in the same manner until leaf nodes are created where the number

of samples in the node is less than a user-defined value. The tree structure, splitting criteria, and samples at each node are stored for use in the online portion of the Hypervisor system.

Online Estimation and Update

Figure 5 summarizes the behavior of our online estimation and update module. Upon receiving an estimation request from a remote JIST client, the Hypervisor system loads the relevant tree into memory. The system then traverses down the tree by comparing the input vector of the incoming request with the splitting rules stored at each node in the tree. Once the system reaches a leaf node L the system returns an estimate given by

$$\mu(L)+\alpha \times \sigma(L) \quad (2)$$

where a is a user defined scaling factor and $\mu(L)$ and $\sigma(L)$ are the mean and standard deviation of the output values of the feature vectors stored at that leaf node. In effect, the user selects an a value that dictates the number of standard deviations away from the mean she is comfortable overestimating; a larger a value would be less likely to fail, but smaller a values may waste less resources. In the simplified case of a Gaussian distribution at a node, this value would be roughly analogous to choosing the likelihood that an estimate would be sufficient for successful execution.

Breiman (Breiman et al. 1984) has pointed out that the sample mean and standard deviation at a node are generally poor estimators of the true mean and standard deviations for that particular cluster. We prefer to continually refine our estimates until the next offline training occurs. To address both of these issues, we use the incoming results of successfully executed JIST processes to update the tree estimates. Upon successful completion of a JIST process, the input and output results are sent to the Hypervisor. The system then uses the input data to traverse down the tree to a leaf in the same manner as the estimation phase. However, this time, the new output is added to the outputs stored in the leaf node. This procedure then has the benefit of quickly updating the mean and standard deviation at the leaf node in order to provide better values for future requests.

Experimental Design

One commonly used layout (“Siemens_CATNAP” (Li et al. 2012)) for JIST was run approximately 1,000 times on 42 different datasets across 22 machines and our local high performance computing cluster. During data collection, memory and time limits were set such that no jobs failed. Table 1 provides a breakdown of the overall scope of the computing resources utilized for generating resource usage data. We used 300 randomly selected data points to train the CPU time and memory decision trees for each algorithm. CPU trees were set to a leaf size of 50 and a of 4 while the memory trees were set to a leaf size of 50 and a of 2. Such testing parameters are arbitrary; we chose tree parameters that provided reasonable estimates from our initial experiments. To provide a baseline comparison, we consider two simple, but highly conservative estimators that would most likely be employed by a human user when attempting to estimate the resource usage of his JIST layout. The first, which we call “Worst Seen FIFO”, uses the worst-case resource requirement observed thus far in the randomly ordered set of samples. Random ordering was used to ensure that

unrecognized structure in job calls did not influence outcomes. To initialize Worst Seen FIFO, a very large estimate is used with the first job so that an initial worst case may be observed. We also consider a "Worst case Omniscient" estimator that has a priori knowledge of the worst-case resource usage of all the jobs in a set and always returned that value as its estimate.

Results

We examined two primary concerns of a resource estimation system: percentage of jobs failed and average overestimation error. A "failed job" indicates that the prediction given by the system was less than the actual resources needed and the job would have been terminated. To quantify excess resources, we find the overestimation error as the ratio of the system's prediction to the actual required resources. Tables 2 and 3 summarize the performance of all three estimators with respect to the failure rates and the overestimation errors.

To concisely summarize performance, we consider the total amount of wasted resources (i.e., reserved, but not used). This metric includes all of the resources that were wasted due to failed jobs in addition to the unused resources allocated due to overestimation. As a simple model we consider a resource estimate $\hat{y}$. Every time the job fails due to an underestimate, we increase the estimate by a unit multiplicative factor (i.e., 2×, 3×, 4×, …). Therefore, if it takes *a* attempts to get an estimate that is sufficient for execution, the person has effectively allocated $1\hat{y}+2\hat{y}+\ldots+(a-1)\hat{y}+a\hat{y}=\frac{a(a+1)}{2}\hat{y}$ resources. If the job actually takes *y* resources, we define the total amount of resources wasted as

$$\frac{\alpha(\alpha+1)}{2}\hat{y}-y \quad (3)$$

Figure 6 shows the results of the three estimators with respect to the different algorithms in our dataset.

Sensitivity to Training Parameters

For our offline training module, we examined the effect of the control parameters on the quality of the produced decision trees in terms of total wasted resources. First, we used 300 data points to train the trees while keeping the alpha value set to 4 for CPU estimates and 2 for memory estimates and tested, in increments of 10, varying leaf sizes from 10 to 300. Figure 7 shows the total wasted resources for each algorithm for both the CPU and memory estimator along with the line styles used for each of the different algorithms. Second, we again trained each tree on 300 data points while keeping the leaf size fixed at 50 for each, but varied the alpha values from 0 to 10 (Fig. 8). Finally, to observe the estimator's behavior in a continuous use environment with new data constantly added to the database, we look at the resources wasted as the training size increases, in increments of 20, from 20 samples to 500 samples. For a training size of *s* samples, the leaf size would be set to $\sqrt{s}$. Figure 9 shows how the estimator's performance scales with the training size.

Discussion

The presented Hypervisor resource estimation framework strikes a good balance between acceptable failure rates and overestimation errors (Tables 1–3). While the failure rates of the FIFO estimator are lower, the presented method is able to provide much tighter CPU time estimates that waste significantly fewer resources at the cost of only a slight gain in the overall percentage of failed jobs (even when including the overhead associated with restarting failed jobs). Hence, over the course of a large study, it is better to allow a few jobs to fail (and be resubmitted) so that more jobs may start more quickly and place resource demands on shared systems. The same conclusion can be drawn for the memory estimator even though the results of individual algorithms are more mixed. Upon examination of the processing modules with the worst resource estimation performance, we see that these modules relied more on the actual content of the input files rather than the size of the files themselves. Such algorithms highlight a unique difficulty of performing resource estimation in a medical context — the need to collect context specific data while respecting patient privacy (i.e., we do not transmit identifiable information or any actual image data to the Hypervisor). Nonetheless, the Hypervisor system is capable of providing estimates that are competitive, if not better, than the fixed-value controls that could be using without an online estimation system.

For both the CPU and memory estimators (Fig. 7), there appears to be little effect of leaf size on resource waste. However, caution should be given for leaf sizes that are too large as the estimator loses its clustering capabilities and provides worse estimates by converging to a running average estimator. On the other hand, we recommend against choosing too small of leaf sizes as one would run the risk of over-fitting the data.

The optimal a seen in Fig. 8 depends on the underlying algorithm. On the whole however, the alpha value appears to reflect its intended function in the estimation algorithm's behavior; a smaller alpha value roughly relates to more wasted resources since a particular module would be more likely to fail. On the other hand, the overall shape of the graphs show that, at a certain point, the alpha value has much more of an effect in increasing the overestimation error than decreasing the failure rate. Consistent with this model, intermediate values of alpha would be optimal and we find that alpha values between 2 and 5 work well in practice to balance the tradeoff between failure rates and overestimation errors.

For the CPU estimator, the Hypervisor provides consistent results regardless of the number of data points available for training (Fig. 9). The memory estimator depicts a similar behavior; yet there seems to be a higher sensitivity to training size for some of the algorithms. On the whole however, the estimator appears to scale well with incoming data by reliably supplying the same level of performance.

The proposed system is modular. First, the backend runs on an Apache Axis 2 server (available on the "Hypervisor" module of the NITRC project JIST). The communication protocol follows Axis 2 standards and is not JIST dependent. The information captured is the algorithm name and high-level descriptions of the input data. Our research group runs

the Hypervisor as a software-as-a-service (SAAS) on a virtual Linux server. These resources can be accessed with any Axis 2 compliant interface. Second, we have constructed an Axis 2 interface for JIST that reports data to Axis 2 and retrieves estimates. This code runs on the user's machine and could be directed to another monitoring service via a user-specified IP address (if a developer wished to develop another Hypervisor implementation or deployment).

In summary, we have demonstrated an implementation for a resource estimation system for use in medical image computing. While future work would involve investigations into more complex machine learning methods and more detailed data collection, we have shown reasonable resource estimates are attainable using straightforward methods from the resource estimation community. The Hypervisor framework enables medical imaging users to more effectively leverage high performance computing environments.

Acknowledgments

This research was supported in part by 1R03EB012461 and R01EB15611. This work was conducted in part using the resources of the Advanced Computing Center for Research and Education at Vanderbilt University, Nashville, TN. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH.

References

Breiman, L. Classification and regression trees. Belmont: Wadsworth International Group; 1984.

Briand LC, Wieczorek I. Resource estimation in software engineering. Encyclopedia of Software Engineering. 2002

Covington, K. MS Thesis. Vanderbilt University; 2011. Informatics for high-throughput and distrubted analysis of medical images.

Dejaeger K, Verbeke W, Martens D, Baesens B. Data mining techniques for software effort estimation: a comparative study. Software Engineering, IEEE Transactions on. 2012; 38:375–397.

Gorgolewski K, Burns CD, Madison C, Clark D, Halchenko YO, Waskom ML, et al. Nipype: a flexible, lightweight and extensible neuroimaging data processing framework in python. Frontiers in Neuroinformatics. 2011; 5

Gray AR, MacDonell SG. A comparison of techniques for developing predictive models of software metrics. Information and Software Technology. 1997; 39:425–437.

Iverson MA, Ozguner F, Potter LC. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. 1999:99–111.

Konstantinides K, Rasure J. The Khoros software development environment for image and signal processing. Image Processing, IEEE Transactions on. 1994; 3:243–252.

Li B, Bryan F, Landman B. Next generation of the JAVA Image Science Toolkit (JIST) visualization and validation. InSight Journal. 2012; 08

Lucas B, Abram G, Collins N, Epstein D, Gresh D, McAuliffe K, et al. An architecture for a scientific visualization system. 1992:107–114.

Lucas B, Bogovic J, Carass A, Bazin PL, Prince J, Pham D, et al. The Java Image Science Toolkit (JIST) for rapid prototyping and publishing of neuroimaging software. Neuroinformatics. 2010; 8:5–17. [PubMed: 20077162]

Marcus, DS.; Olsen, TR.; Ramaratnam, M.; Buckner, RL. XNAT: a software framework for manging neuroimaging laboratory data. Ontario: Organization of Human Brain Mapping; 2005.

McAuliffe MJ, Lalonde FM, McGarry D, Gandler W, Csaky K, Trus BL. Medical image processing, analysis & visualization in clinical research. Computer-Based Medical Systems, IEEE Symposium on. 2001; 0:381.

Parker, SG.; Johnson, CR. SCIRun: a scientific programming environment for computational steering. Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM); ACM; 1995.

Pieper, S.; Lorensen, B.; Schroeder, W.; Kikinis, R. The NA-MIC Kit: ITK, VTK, pipelines, grids and 3D slicer as an open platform for the medical image computing community. Biomedical Imaging: Nano to Macro, 2006. 3rd IEEE International Symposium on; 2006. p. 698-701.

Rex DE, Ma JQ, Toga AW. The LONI pipeline processing environment. NeuroImage. 2003; 19:1033–1048. [PubMed: 12880830]

Sheehan B, Fuller S, Pique M, Yeager M. AVS software for visualization in molecular microscopy. Journal of Structural Biology. 1996; 116:99–106. [PubMed: 8742730]

Smith, W.; Foster, I.; Taylor, V. Job Scheduling Strategies for Parallel Processing. London: Springer; 1998. Predicting application run times using historical information.

Sonmez, O.; Yigitbasi, N.; Iosup, A.; Epema, D. Trace-based evaluation of job runtime and queue wait time predictions in grids. Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing; ACM; 2009.

Turing A. On computable numbers, with an application to the Entscheidungsproblem (1936). B Jack Copeland. 2004:58.

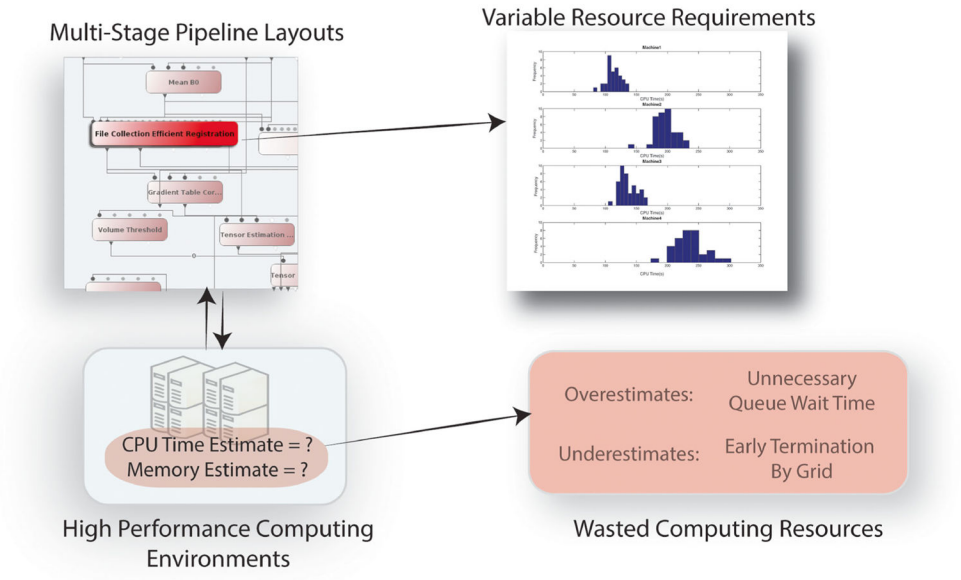

Fig. 1.

Overview of the need for accurate estimates in high performance medical imaging. Complex multi-stage imaging layouts need to provide accurate estimations of their resource requirements in order to leverage high performance computing resources. The upper-right inlay shows histograms of the runtime for a single software module ("File Collection Efficient Registration") on datasets acquired on 21 individuals

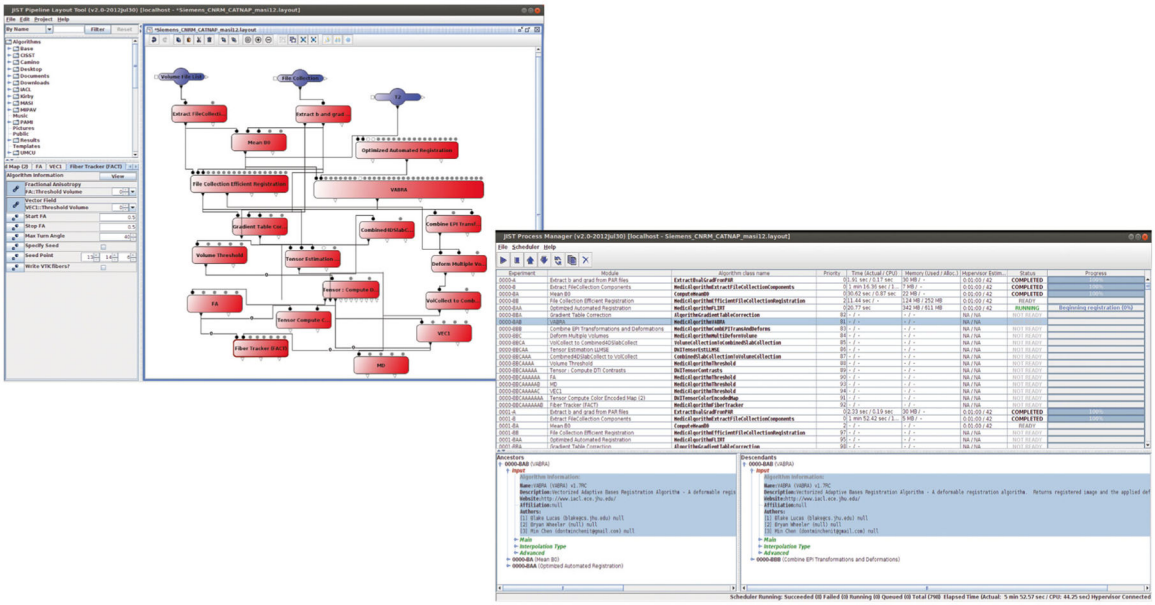

Fig. 2.

User interface for the JIST system. The JIST Layout Tool (Left) provides visual programing capabilities for medical imaging pipelines while the JIST Process Manager (Right) manages dependencies and controls job execution

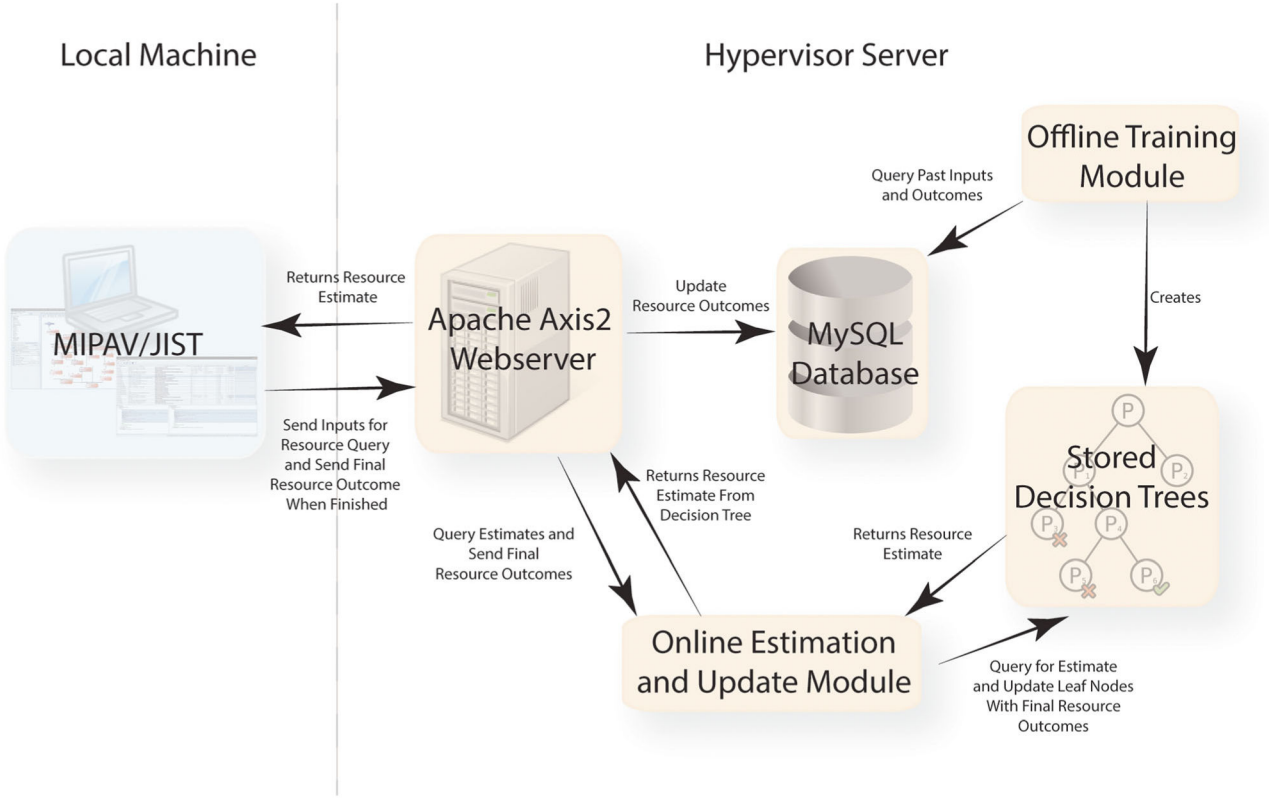

Fig. 3.

Overview and flowchart of the Hypervisor system. The resource estimation system can be enabled through the JIST Layout Tool; users can click to opt whether or not to get a Hypervisor resource estimate and to select the address of a Hypervisor server to use. By default, a public Hypervisor is specified. Once the user wishes to run a created layout by running the JIST Process Manager, resource estimates will be automatically available to JIST experiments marked as “Ready”. JIST will send to the Hypervisor an estimation request along with the following information: the name of the process, the user’s hardware environment, and the inputs of the process. In the interest of privacy, no files or filenames are collected—file sizes are sent to the Hypervisor instead. Furthermore, upon the actual execution of a particular JIST process, another estimate will be requested from the Hypervisor server. Once the JIST process completes successfully, the same data from above will be saved to the MySQL database along with the final CPU time and memory usage of the task. The online estimation procedure returns resource estimates in several milliseconds while the offline training module runs nightly and in parallel to any resource estimate requests

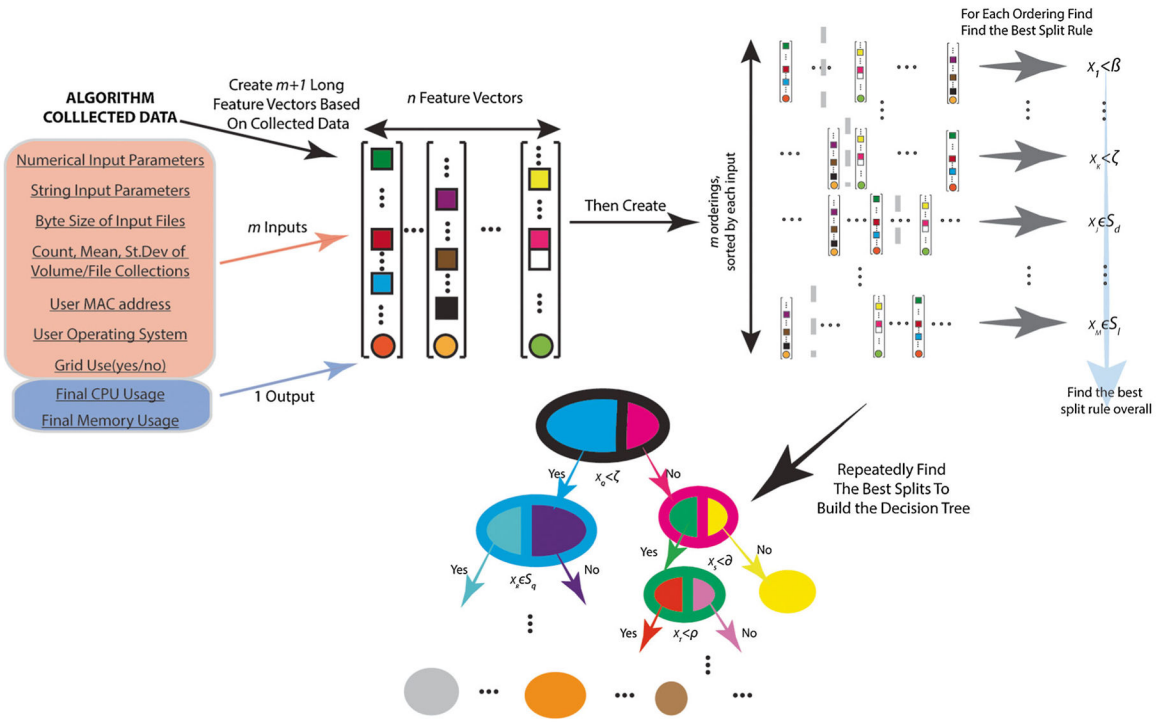

Fig. 4.

Overview of the offline training module of the Hypervisor system. The algorithm creates feature vectors based on the collected input/output data and uses them to train a decision tree. At each node of the decision tree, the algorithm creates child nodes by finding the best individual feature to split the vectors along

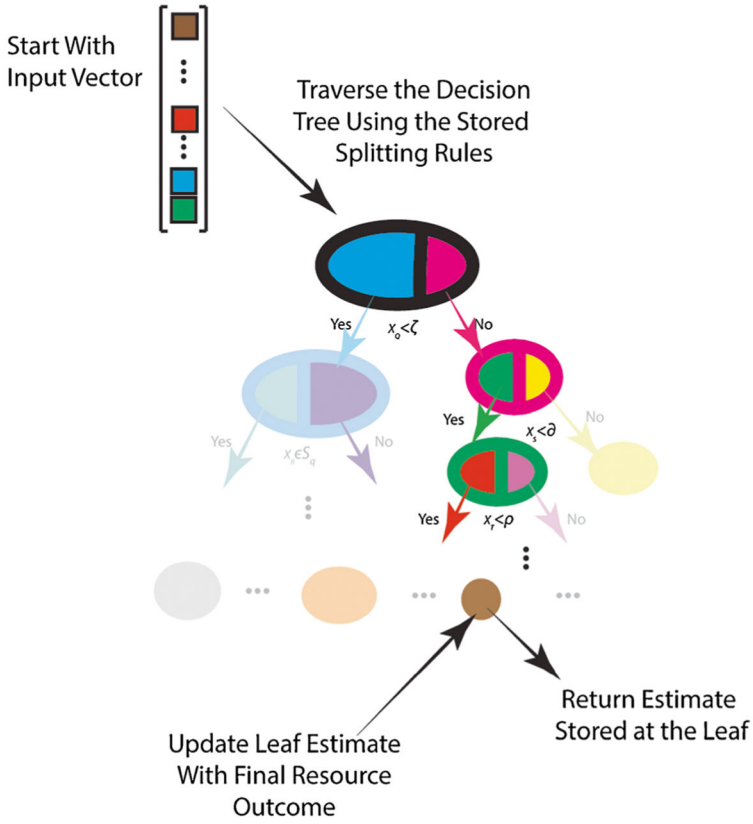

Fig. 5.

Overview of the online estimate and update module. The prediction returned for a particular input vector is given by traversing the tree to a terminal node by using the stored splitting rules found in the offline training portion of the algorithm. That particular terminal node's prediction value is then updated with the actual resource requirements of the current job

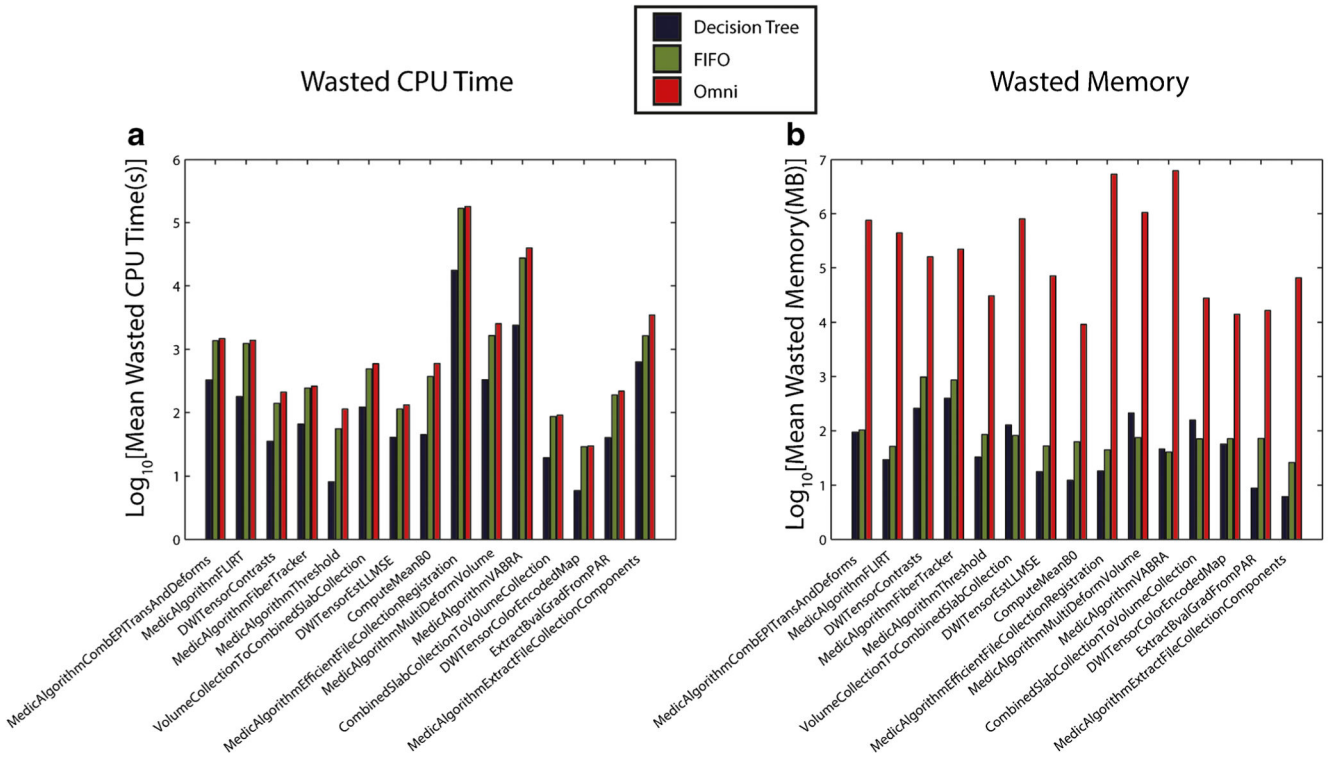

Fig. 6.

Total resources wasted by each estimator, broken down by JIST algorithm

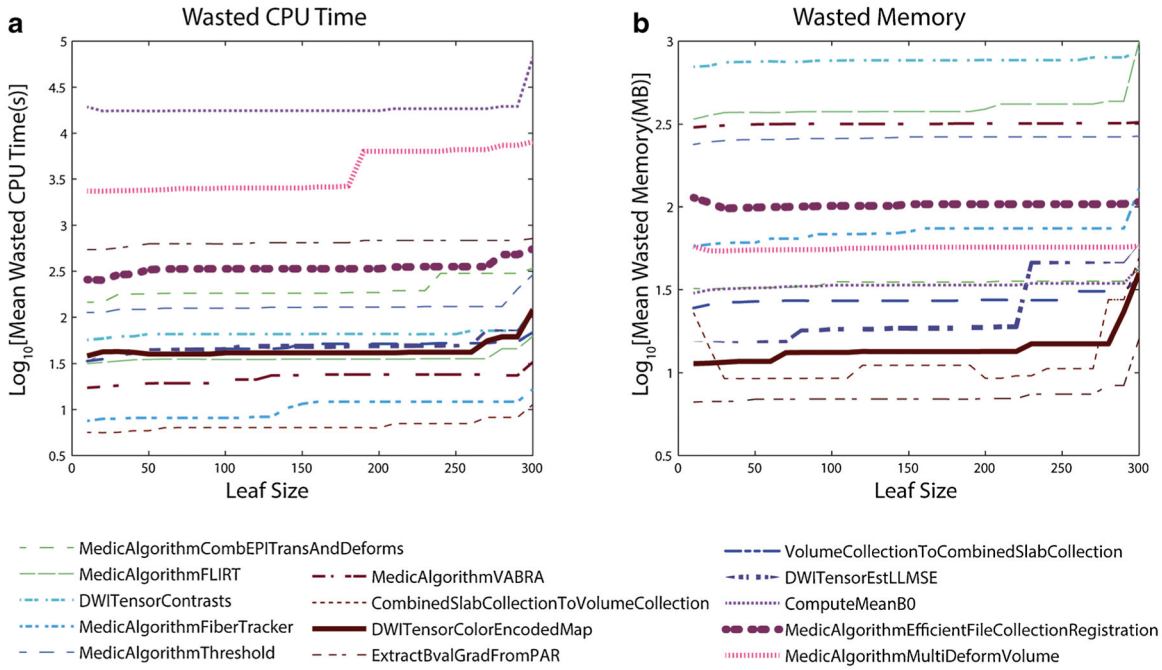

Fig. 7.

Effect of leaf size on the resource waste consumption of our estimator. The training sample was set to 300 with the alpha value for the CPU time estimator and memory estimator set to 4 and 2 respectively

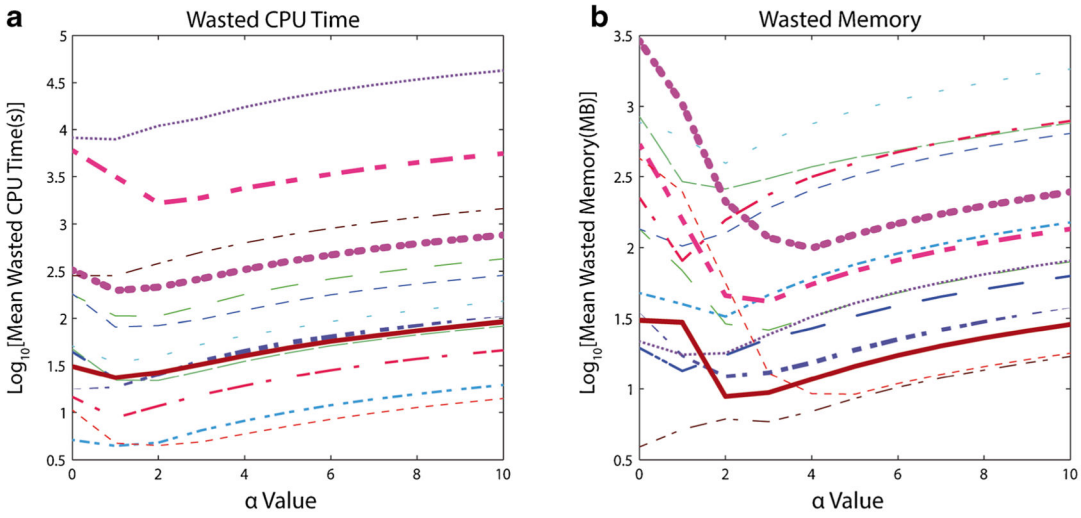

Fig. 8.

Effect of *a* on the resource waste consumption of our estimator. The training sample size was set to 300 with a terminal leaf size of 50 for both the CPU and memory estimators

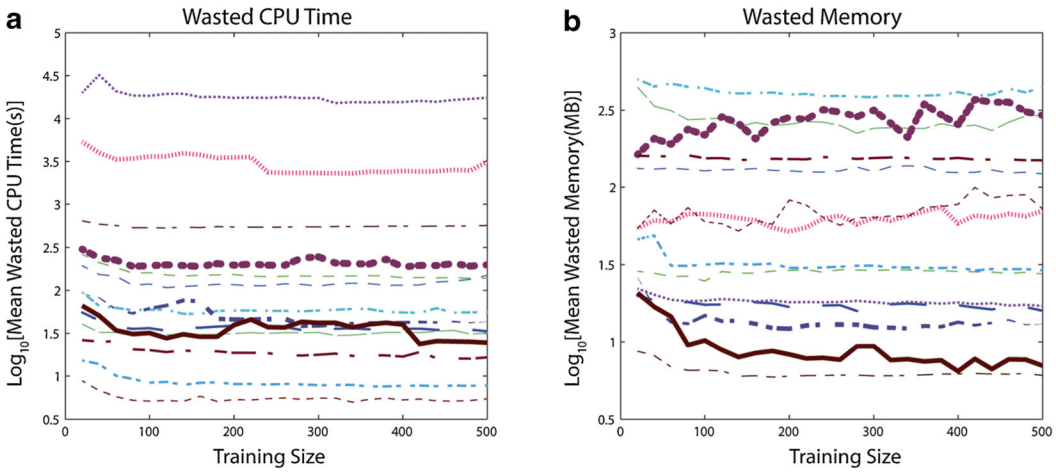

Fig 9.

Overall wasted resource consumption of our algorithm as a function of sample training size. The leaf size was set to vary with the square root of the training sample while the alpha value for the CPU time estimator and the memory estimator was set to 4 and 2 respectively

Table 1

Summary of the data

Algorithm Name	CPU time μ±σ (min)	Memory Used μ±σ (MB)	Number of Data Points	Number of Computers	Total CPU time (Days)
MedicAlgorithmMultiDeformVolume	5.94±2.04	3119.77±26.08	886	20	3.65
DWITensorContrasts	1±0.24	1494.15±247.14	982	20	0.68
ExtractBvalGradFromPAR	0.15±0.47	42.52±9.71	1186	22	0.12
MedicAlgorithmFiberTracker	1.04±0.51	811.99±216.13	980	20	0.71
MedicAlgorithmCombEPITransAndDeforms	13.8±3.43	207.64±54.37	931	21	8.92
MedicAlgorithmVABRA	101.97±31.43	528.73±15	998	22	70.67
DWITensorColorEncodedMap	0.21±0.04	618.16±10.27	981	20	0.14
VolumeCollectionToCombinedSlabCollection	3.33±1.12	100.39±67.35	950	20	2.2
MedicAlgorithmExtractFileCollectionComponents	2.43±3.28	8.86±3.74	1170	22	1.97
MedicAlgorithmThreshold	0.08±0.07	58.06±32.26	3910	20	0.24
ComputeMeanB0	0.2±0.49	44.95±15.21	1149	22	0.16
CombinedSlabCollectionToVolumeCollection	0.31±0.12	151.9±80.96	982	20	0.21
MedicAlgorithmEfficientFileCollectionRegistration	93.04±242.28	20.37±10.9	1078	22	69.65
DWITensorEstLLMSE	0.87±0.26	26.57±9.45	978	20	0.59
MedicAlgorithmFLIRT	3.22±1.27	158.63±9.8	1140	22	2.55
Summary:	15.17 (Avg)	492.85 (Avg)	1220 (Avg)	21 (Avg)	162.46 (Total)

Table 2

CPU usage performance summary

	Baseline Settings: Leaf Size=50, Alpha=0	Recommended Settings: Leaf Size=50, Alpha=4		"Worst Seen FIFO" Estimator		Worst Seen Omnicient" Estimator
Algorithm Name	MSE	†	‡	†	‡	‡
MedicAlgorithmMultiDeformVolume	2.10E+10	0.95	1.48	0.68	4.97	8.54
DWITensorContrasts	6.26E+09	1.19	1.83	0.71	2.99	4.70
ExtractBvalGradFromPAR	9.03E+07	0.44	1.64	1.60	99.34	123.45
MedicAlgorithmFiberTracker	2.60E+08	1.76	1.75	0.71	5.67	5.87
MedicAlgorithmCombEPITransAndDeforms	1.00E+07	2.16	3.17	0.64	2.91	3.08
MedicAlgorithmVABRA	3.24E+09	0.62	2.80	0.80	5.22	7.83
DWITensorColorEncodedMap	2.38E+08	1.92	1.89	0.51	3.37	3.45
VolumeCollectionToCombinedSlabCollection	2.91E+08	2.59	6.38	0.95	4.36	5.00
MedicAlgorithmExtractFileCollectionComponents	1.60E+14	1.41	5.15	0.60	31.84	39.74
MedicAlgorithmThreshold	1.57E+10	1.19	1.54	0.28	16.58	30.61
ComputeMeanB0	2.82E+12	0.29	1.59	0.61	78.10	99.51
CombinedSlabCollectionToVolumeCollection	4.23E+07	1.03	2.05	0.51	5.87	6.29
MedicAlgorithmEfficientFileCollectionRegistration	6.73E+06	1.47	1.49	0.46	43.41	45.08
DWITensorEstLLMSE	2.62E+08	2.93	12.37	1.02	3.49	3.74
MedicAlgorithmFLIRT	4.25E+10	3.45	7.42	0.53	8.44	8.88
Averages:	1.09E+13	1.56	3.50	0.71	21.10	26.38

† Percent Failed

‡ Average Overestimate Error (Predicted/Actual)

Table 3

Memory usage performance summary

	Baseline Settings: Leaf Size=50, Alpha=0	Recommended Settings: Leaf Size=50, Alpha=2		"Worst Seen FIFO" Estimator		"Worst Seen Omnicient" Estimator
Algorithm Name	MSE	†	‡	†	‡	‡
MedicAlgorithmMultiDeformVolume	2383.71	0.79	1.68	0.68	1.02	1.02
DWITensorContrasts	82.71	4.17	1.11	0.71	1.65	1.69
ExtractBvalGradFromPAR	14624.73	3.37	1.11	0.42	2.86	2.96
MedicAlgorithmFiberTracker	40443.20	2.21	1.54	0.92	2.04	2.23
MedicAlgorithmCombEPITransAndDeforms	334.74	1.94	1.59	0.97	1.76	1.79
MedicAlgorithmVABRA	4638.79	0.15	6.63	0.90	1.07	1.07
DWITensorColorEncodedMap	63.62	5.31	1.75	0.41	1.11	1.11
VolumeCollectionToCombinedSlabCollection	93.40	4.12	1.20	0.63	5.31	5.42
MedicAlgorithmExtractFileCollectionComponents	93.08	1.80	2.12	0.51	4.32	5.55
MedicAlgorithmThreshold	745.40	1.88	1.02	0.13	3.25	3.39
ComputeMeanB0	181.79	2.58	1.05	0.35	2.69	2.73
CombinedSlabCollectionToVolumeCollection	7005.94	0.15	19.50	0.41	13.70	13.69
MedicAlgorithmEfficientFileCollectionRegistration	13.57	3.38	1.01	0.56	3.92	4.41
DWITensorEstLLMSE	72.08	1.13	1.12	0.51	5.13	5.59
MedicAlgorithmFLIRT	7.45	9.20	1.49	0.44	1.32	1.36
Averages:	4.72E+03	2.81	2.93	0.57	3.41	3.60

† Percent Failed

‡ Average Overestimate Error (Predicted/Actual)