

Designing and Parameterizing a Workflow for Optimization: A Case Study in Biomedical Imaging

Vijay S. Kumar², Mary Hall¹, Jihie Kim¹, Yolanda Gil¹,
Tahsin Kurc², Ewa Deelman¹, Varun Ratnakar¹, Joel Saltz²

¹University of Southern California
Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
{mhall,jihie,gil,deelman,varunr}@isi.edu

² Department of Biomedical Informatics
The Ohio State University
333 West 10th Avenue
Columbus, OH 43210
{vijayskumar,kurc,saltz}@bmi.osu.edu

Abstract

This paper describes our experience to date employing the systematic mapping and optimization of large-scale scientific application workflows to current and future parallel platforms. The overall goal of the project is to integrate a set of system layers – application program, compiler, run-time environment, knowledge representation, optimization framework, and workflow manager – and through a systematic strategy for workflow mapping, our approach will exploit the vast machine resources available in such parallel platforms to dramatically increase the productivity of application programmers. In this paper, we describe the representation of a biomedical imaging application as a workflow, our early experiences in integrating the set of tools brought together for this project, and implications for future applications.

1 Introduction and Overview

Scientists are conducting data analysis of unprecedented complexity and scale. Many scientific applications are being built not as monolithic entities, but rather by combining models and analysis routines contributed by many scientists specializing in different areas of the problem. Resulting applications can be defined as workflows composed of hundreds or thousands of components to be executed in coordination on a variety of resources.

Efficient, robust execution of application workflows in heterogeneous, distributed environments is composed of a set of problems at different scales—from low-

level architecture-specific optimizations to utilize the memory hierarchy and individual processor, effective multi-core parallelization, on up to high-level application composition. In our project, we view application optimization as hierarchical, consisting of optimization of individual components, and the composition of components into a workflow. Each component comprising a workflow must be able to execute efficiently on the target architecture(s), and under a variety of execution environment conditions, such as resource constraints and data set characteristics. In turn, these performance metrics depend on a variety of application-level features and the set of transformations applied by the compiler. Further, the components must be composed in such a way that the solution performs well globally and makes productive use of valuable computing resources.

To address this complex workflow mapping problem, we have assembled a strategy and a team of experts with existing tools that can be leveraged to automatically map and optimize complex workflows for execution on current and future large-scale computing platforms [1]. In this initial phase of the project, we have focused on integration of the concepts and early prototypes of relevant technologies. We selected a specification application from biomedical imaging to provide a representative workflow for inspiring necessary extensions to the tools and their integration to meet the ambitious goals of this project.

In the remainder of the paper, Section 2 will discuss the four systems that provide the context for this project. Section 3 describes the biomedical imaging application and its representation as a workflow. The exercise of rewriting this application as a workflow and

using the existing tools uncovered a set of issues, some of which have already led to changes in the tools, as discussed in Section 4.

2 Overview of System

Workflows expressed in WINGS. Wings[6] takes in high-level representations of workflows and automatically generates detailed specifications of the tasks and dataflow for submission to the Pegasus mapping and execution system. Wings represents workflows using semantic metadata properties of both components and datasets, represented in OWL. Using these high-level and semantically rich representations, Wings generates the details that Pegasus needs to map the workflows to the execution environment, and to generate metadata descriptions of new datasets that result from workflow execution.

Scheduling and mapping of Workflows in Pegasus. The Pegasus Workflow Management System (Pegasus-WMS) consists of the Pegasus mapper [5] and the DAGMan workflow execution engine. Pegasus takes a workflow description generated by Wings in the form of a Directed Acyclic Graph in XML format (DAX). Pegasus can map workflows onto a variety of target resources such as those managed by PBS, LSF, Condor, and individual machines. The executable workflow produced by the Pegasus mapper has directives to DAGMan for the execution of the workflow components. These directives include remote job execution, data movement, and data registration.

Managing data in DataCutter. The DataCutter component-based middleware framework provides a coarse-grained data-flow system and allows combined use of task- and data-parallelism [2]. In DataCutter, application processing is implemented as a set of components, referred to as *filters*, that exchange data through a *stream* abstraction. A stream denotes a unidirectional data flow from one filter (i.e., the producer) to another (i.e., the consumer). The DataCutter runtime system supports execution of filters on heterogeneous collections of storage and compute clusters in a distributed environment. Multiple copies of a filter can be created and executed, resulting in data parallelism.

Model-guided empirical optimization in ECO. The Empirical Compilation and Optimization (ECO) compiler uses an approach to compiler-directed code optimization called *model-guided empirical optimization* [3]. This approach simultaneously optimizes across multiple levels of the memory hierarchy for dense-matrix computations by combining compiler models and heuristics with guided empirical search. In

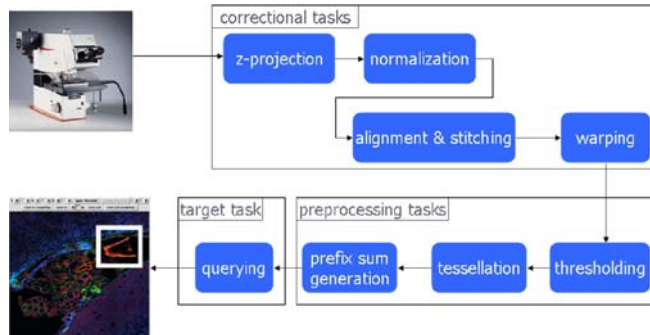


Figure 1. Overview of Larger Pipeline

the proposed system, components for which this completely compiler-based optimization strategy is applicable are tuned automatically for multiple levels of the memory hierarchy and multi-core processors. Combining this system with recent research on how to generate optimized code with application-level parameters for a molecular dynamics application [1] will enable us to explore a wide range of optimization strategies for workflows.

3 Case Study: Biomedical Imaging

3.1 Description of Workflow

Figure 1 shows a pipeline developed by neuroscientists to study signal distribution in mouse brain tissue samples (digitized using confocal microscopes). When digitizing the tissue at high resolutions, the limited field-of-view (FOV) of the microscopy instrument comes into play. Hence, the image data is acquired as a collection of smaller rectangular **tiles**, each containing high resolution data corresponding to a given FOV.

During image acquisition, a microscope moves a sensor in X and Y directions in fixed steps. At each step, multiple images at different focal lengths (Z’s) are captured. Each such image corresponds to a **tile** in the entire image. Neighbor tiles overlap each other to some extent. At the end of the acquisition, an image consists of many tiles stored in a single file, usually with multiple Z slices. Each slice corresponds to a different focal plane. Prior to any analysis, these tiles need to be corrected for artifacts and stitched to produce a complete mosaic of image data.

In this work, we focus on part of the *correctional tasks* stage of the pipeline. This stage consists of processing steps on the tile data at the end of which we have a “stitched” image montage. Step 1 is called *Declustering* and involves conversion of image data from any input format (such as JPEG, TIFF, PPM or other proprietary image formats) into a single common format for distributed storage and analysis across

a cluster of compute nodes. The declustering stage also partitions the image data into smaller portions and distributes them across nodes in some predefined manner, as the image datasets that are potentially hundreds of Gigabytes in size.

Image Data Representation. A tile contains pixels in a rectangular subregion of a single focal plane. It is associated with a bounding box in X and Y dimensions of the image and a Z value that specifies the focal plane of the tile. To minimize inter-processor communication and disk I/O, we group neighboring tiles to form **chunks**. A chunk consists of a subset of tiles that are neighbors in X and Y dimensions of the image, and a Z value that specifies the focal plane of the tile. A chunk provides a higher-level abstraction for data distribution, the unit of i/o. A tile is assigned to a single chunk, and a chunk is stored on only one node. A node may store multiple chunks. A set of chunks that share the same X and Y limits but differ only in their Z values are said to form a **stack** of chunks. The set of all chunks that share the same Z value form a **slice** of chunks.

The remaining steps of the pipeline are as follows: a *Z projection* is performed on the image (Step 2). This step collapses all Z slices in the input image into a single output slice using a *max* operator. The next step is a *normalization* operation (Step 3) that corrects for lighting variances across tiles. This step takes the Z projected slice from Step 2 as input and computes the average and offset intensity tiles for that slice. These tiles are then used to normalize the image data. The Z projected slice is normalized first, followed by each slice of the original image.

Next is the *auto-alignment* operation (Step 4), which determines how the partially overlapping tiles within a slice should converge upon one another to produce a properly aligned slice, eliminating all the overlapping regions. This step determines the “best match” between every pair of local adjacent tiles in the normalized Z projected slice. A spanning tree algorithm is used to determine the overall best global alignment from these local alignments. This step produces a subset of the input offsets which define the exact positions of each tile in the final montage. The last step is *Stitching*(Step 5) that processes these displacements, moves the tile data and produces the stitched image.

A detailed explanation of the image processing pipeline and its distributed implementation is provided in [4, 7].

Performance Parameters. The chunking strategy used plays a critical role with respect to the partitioning and storage of the image data. Hence, the dimen-

sions (shape and size) of the chunks must be chosen very carefully as they have huge performance implications. Since a chunk is a unit of I/O, it directly impacts the amount of disk I/O and inter-processor communication. While choosing a set of chunk dimensions, one must take into account the characteristics of image data (size of the image, number of focal planes, size of a tile, distribution of pixel density and so on), the resource characteristics (number of nodes, available disk capacity, memory on the nodes, communication fabric, processor speeds and so on) and the data access and communication patterns of the image processing operations. In some cases, an added dimension to consider would be the quality-of-service(QoS) requirements put forward by the user. Moreover, we are addressing pipelines of operations; the right chunk dimensions for one stage may not be optimal for other stages. We wish to obtain chunk dimensions that are globally optimal for the end-to-end workflow.

3.2 WINGS Workflow Representation

In this section, we highlight our efforts and issues faced in representing the workflow from Section 3.1 in WINGS, including required changes to the application to convert it into a WINGS workflow. As is the case for any new application workflow, a domain specific file ontology and component library were created for this application.

Our initial representation corresponded exactly to the description of the workflow provided in Section 3.1. For every step in the workflow description, we created a component in the WINGS workflow. Each component acts as a placeholder for the actual task being performed in that step of the workflow.

Lessons Learned. The following three observations from this initial workflow representation pointed to required changes to accomplish our optimization goals.

Code modifications: The controllable performance parameter (in this case, chunk size) was not an explicit parameter exposed to WINGS, but rather part of the metadata within the *decluster* stage.

Finer granularity of components: The initial workflow instance essentially consisted of a sequence of DataCutter jobs (one job per component) that would be executed via Pegasus. The absence of Component Collections in the workflow meant that all parallelism was to be handled implicitly within the components rather than exposed to WINGS.

Complete instantiation of workflow template: Data flow was only loosely described in terms of metadata (header files) being exchanged between the steps, as opposed to references to real data. What was needed

was a description of the relationship between different portions of the image that a component processed.

Modified Workflow. On account of these observations, we developed a modified workflow that exposed the optimization opportunities to the system.

Expose Controllable Performance Parameter: We divided the decluster step from the initial representation into two stages: *Format Conversion* and *Chunkify*. Format conversion is responsible for conversion of image data from any input format into a common distributed image format for analysis. The image data is extracted and read into internal data structures, so that the remainder of the workflow is independent of the input image format. The chunkify stage now explicitly takes as input a performance parameter that is controlled by the system. Based on the chunking strategy chosen by the system, one can determine the **number of chunks in an image slice**, p . Using image data characteristics from the output of the format conversion stage and resource characteristics from Pegasus, the system decides on a suitable chunking strategy to be adopted for the workflow. Through WINGS, the performance parameter can be exposed and the value adjusted until the required performance is attained.

Adjust Granularity of Mapping Choices: Through analysis of the workflow, we identified ways to modify the WINGS ontology for the workflow into a set of finer-grained mapping decisions, as compared to the previous workflow. For example, the Z projection operation is a pleasingly parallelizable operation that can be performed on each stack of chunks independently. So, given a value p for the performance parameter, there will exist p stacks of chunks for the entire image. In this way, the Z projection task can be represented as a collection of p independent jobs, where each job involves a Z projection operation on a stack of chunks. We also separate the normalization of the Z projected slice from that of the chunkified image, and represent the former as a new component (*normalizeZ*) that can be executed concurrently.

We split components into explicit sub-components, where it increases the potential for parallel execution. The normalization operation for an image is split into two sub-components: the *prenormalize* step that generates the average and offset tiles and the actual *normalize* step that uses these tiles to normalize the data. Now, the latter is again pleasingly parallelizable and is represented as a collection of p jobs in WINGS. Similarly, the alignment operation was split into a local *autoalign* step and a global *mst* step that executes the spanning tree algorithm.

After the above changes, the nature of the compo-

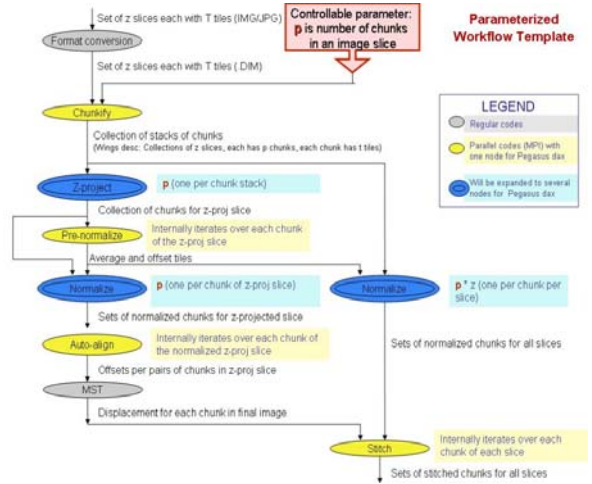


Figure 2. WINGS representation of workflow.

nents was exposed. We identified *format conversion*, *chunkify* and *mst* as simple jobs that execute on a single node. The *zproject*, *normalizeZ* and *normalize* components are collections consisting of p independently executable jobs. Lastly, the *prenormalize*, *autoalign* and *stitch* jobs are MPI-style parallel codes that need inter-processor communication.

Instantiating the Template with Data Files: We expressed the relationship between different data files directly in WINGS, so that the data could always be colocated with its computation.

The above changes were incorporated to the WINGS domain-specific ontologies along with appropriate data types. The new representation of the workflow is as shown in the workflow template in Figure 2.

Following modifications to the workflow representation, the system can now rely on Pegasus to choose a set of available compute nodes to execute the workflow. Previously, the user provided a host file containing compute nodes for execution and the locations on disk of these nodes for data storage.

4 Current and Future Work: Impact on Existing Tools

Adding tunable parameters to WINGS. To support tunable parameters, we extended WINGS to include variables in the mapping function that are bound at the time of workflow execution. These parameters can be included in the specification of a workflow component that splits an initial dataset into smaller ones, and control the number and size of results produced by the component.

In the case of the biomedical imaging workflow, the tunable parameter is the chunksize, and the appropriate chunksize depends on features of the execution en-

environment such as available storage and computational capability. Consider how WINGS might use feedback from the execution environment as it makes decisions about unrolling parallel computations. In particular, consider the number of hosts available for execution. If Wings was aware that k hosts were available for computation, it should target to divide the dataset into k chunks.

The value of such parameters are currently provided manually, but eventually it will be obtained automatically when Wings queries a service that monitors the available resources and assigns some number of execution resources (as well as other features such as capability of the platform) to the workflow at hand. Wings uses this value to configure the component, which will be set to generate the appropriate amount of chunks. Wings then unrolls the remainder of the workflow according to that value.

Combining Pegasus and DataCutter The Pegasus mapper generates an executable workflow that includes directives for data movement (staging of the input data and software to the execution sites and of results to permanent storage) and data registration, where data location information is saved in a data registry. When looking at the integration with DataCutter, we see an overlap in functionality in the data management aspects of the workflow. For example, DataCutter makes a decision of how to partition the input data based on the availability of the resources and then lays out that data on the nodes of the cluster. The data layout information is then made available to the DataCutter filters that perform the data processing.

In general we see the interactions between Pegasus and DataCutter as complimentary, where they work in concert to manage the data across the computational clusters and storage resources and within the clusters. Ultimately, we will perform an integration of the two technologies, where Pegasus will be able select the target computational sites, stage the data to them and stage the results back to archival storage. Once the data are staged-in Pegasus would invoke DataCutter on the cluster, and then DataCutter would be responsible for the data management across the storage and compute nodes of the cluster and the computations that need to occur on that data. As a result, we will have a hierarchical workflow management system, where Pegasus would manage the data and workflow components (including DataCutter) across the distributed resources and DataCutter would do that management within a collection of storage and compute clusters on a local area network.

Integrating parameter-based optimization. We would like to integrate the tunable parameters in WINGS with the compiler technology, to make it possible to generate at execution time those portions of the code that depend on the value of the tunable parameter. We have extended the ECO compiler to include a script interface that describes the parameterized transformation of a piece of code. Once the parameter value is bound, the parameter in the script can be bound, and the code generated automatically from this bound specification.

Acknowledgements. This research has been supported by the National Science Foundation under awards CSR-0509517 and CSR-0615412.

References

- [1] B. Bansal, U. Catalyurek, J. Chame, C. Chen, E. Deelman, Y. Gil, M. Hall, V. Kumar, T. Kurc, K. Lerman, A. Nakano, Y. Nelson, J. Saltz, A. Sharma, and P. Vashishta. Intelligent optimization of parallel and distributed applications. In *Proceedings of the Workshop on Next Generation Software, held in conjunction with IPDPS '07*, Mar. 2007.
- [2] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with datacutter. *Parallel Computing*, 27(11):1457–1478, Oct. 2001.
- [3] C. Chen, J. Chame, and M. W. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Proceedings of the International Symposium on Code Generation and Optimization*, Mar. 2005.
- [4] S. K. Chow, H. Hakoziaki, D. L. Price, N. A. B. MacLean, T. J. Deerinck, J. C. Bouwer, M. E. Martone, S. T. Peltier, and M. H. Ellisman. Automated microscopy system for mosaic acquisition and processing. *Journal of Microscopy*, 222(2):76–84, May 2006.
- [5] E. Deelman and et al. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13:219–237, 2005.
- [6] Y. Gil, V. Ratnakar, E. Deelman, G. Mehta, and J. Kim. Wings for pegasus: Creating large-scale scientific applications using semantic representations of computational workflows. In *Proceedings of the 19th Annual Conference on Innovative Applications of Artificial Intelligence (IAAI)*, July 2007.
- [7] V. Kumar, B. Rutt, T. Kurc, U. Catalyurek, S. Chow, S. Lamont, M. Martone, and J. Saltz. Large image correction and warping in a cluster environment. In *Proceedings of SC 2006*, Nov. 2006.