Enabling Inter-Machine Parallelism in High-Level Languages with SEJITS and MapReduce

Michael Driscoll
mbdriscoll@cs.berkeley.edu

Evangelos Georganas
egeor@cs.berkeley.edu

Penporn Koanantakool
penpornk@cs.berkeley.edu

Computer Science Division
University of California, Berkeley

ABSTRACT

Selective, embedded, just-in-time specialization (SEJITS) is a technique for optimizing embedded domain-specific languages through the use of *specializers*, or code modules developed by expert programmers that target particular accelerators such as multicore processors and GPUs via just-in-time compilation. We extend SEJITS to exploit inter-machine parallelism by targeting clusters of machines via MapReduce. Our work enables the development of specializers for large, data-parallel applications whose workflows can be cast as MapReduce operations. We present an implementation that targets Hadoop and we describe specializers for two applications. The first, a pure-Python protein docking application, requires a 1-line change to realize a 280x speedup on a cluster with 450 cores. The second, an audio processing application, demonstrates our approach's ability to leverage clusters of GPU-equipped machines by composing parallel programming patterns. Results indicate that clusters are viable targets for specialization, and that pattern composition is a useful technique for managing multi-level parallelism.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Concurrent, distributed, and parallel languages; D.2.11 [**Software Architectures**]: Patterns

General Terms

Languages, Design, Performance

Keywords

mapreduce, dynamic optimization, cloud computing, embedded domain specific languages, parallel programming techniques, scientific computing

1. INTRODUCTION

Computational scientists are often concerned with the time-to-solution for simulations, but they must frequently choose between flexible implementations and efficient ones. The gap between productivity and performance is even greater for applications that require massive parallelism because of the increased complexity of parallel programming. Scientists often lack the expertise to develop massively parallel applications in *efficiency-level languages* like C and Fortran, yet they require the performance of native code. Instead, many scientists prefer *productivity-level languages* like Python that provide high-level abstractions and enable rapid iteration, but whose programs run orders of magnitude more slowly than analogues written in efficiency-level languages. This paper introduces a technique for achieving massive parallelism without sacrificing flexibility through the composition of parallel programming patterns.

One approach to bridging the performance-productivity gap is SEJITS, or Selective, Embedded, Just-In-Time Specialization [11]. SEJITS is a collection of techniques for optimizing applications written in productivity-level languages like Python. The techniques have several attractive properties:

Dynamic. As "JIT" implies, specialization occurs at runtime; therefore, both hardware properties and data characteristics can inform optimization logic.

Selective. Only a pre-defined subset of the productivity-level language (the "host" language) is specialized. The remaining code is executed normally (i.e. by the Python interpreter).

Embedded. The specialized logic is a subset of the host language. When application-level logic cannot be expressed in the specialized subset, the programmer can fall back to host language facilities.

Specialization is managed by code modules called *specializers*. Specializers are implemented by expert programmers and provide mappings from productivity-level code to any number of accelerators through SEJITS "backends". Existing backends target multicore CPUs and GPUs via JIT compilation of Cilk++ and CUDA, respectively. Specializer developers are also encouraged to provide an implementation purely in the productivity-level language in case no accelerators are available or specialization would not be profitable.

Specialization is not limited to targeting CPUs or GPUs via JIT compilation. This paper explores the viability of

an alternate approach: targeting clusters of machines via MapReduce. Such an approach can be realized in a new backend that allows specializer developers to exploit opportunities for inter-machine parallelism by casting application logic as MapReduce workflows and delegating its execution to a cluster of machines. We opted to target MapReduce for two main reasons:

Availability MapReduce, usually realized in Hadoop [2], is widely available on private clusters, shared clusters at national supercomputing centers, and public services such as Amazon Elastic MapReduce [1].

Distributed Computing Features Hadoop provides fault tolerance, data distribution, load balancing, and other features that would be costly to implement and maintain in the backend. It also has a broad set of tuning parameters for configuring the cluster.

The choice of MapReduce as a target limits the set of specializable logic to that which can be cast in a MapReduce workflow. Fortunately, most large, data-parallel applications fall into this class [20]. This choice of MapReduce is not fundamental to specialization: one could imagine targeting clusters for tightly-coupled applications, but such techniques are beyond the scope of this paper.

2. RELATED WORK

Our work embodies the synthesis of three topics: composition of parallel programming patterns, domain-specific language support for inter-machine parallelism, and scientific cloud computing.

Composition of Parallel Programming Patterns.

Several projects have focused on the composition of data-parallel DSLs in high-level, productivity-oriented languages. The Copperhead [10] language provides a set of data-parallel abstractions expressed as a restricted subset of Python. Copperhead targets the nested data-parallel model, wherein parallel operations can be composed to arbitrary depth. The Copperhead compiler can efficiently map the nested model to parallel architectures like GPUs, multicore processors, and vector register machines. Like a MapReduce-backed specializer, Copperhead enables parallelism through its language specification, but its current implementation only exploits intra-machine parallelism.

Data Parallel Haskell [12] targets multicore processors by providing the same nested data-parallelism in a higher-order, functional language.

Lithe [18] enables composition of parallelism by providing basic primitives for cooperatively sharing computational resources between parallel libraries within an application.

Language Support for Inter-Machine Parallelism.

A number of languages provide support for expressing inter-machine parallelism. The MapReduce API itself can be viewed as a domain-specific language embedded in a general purpose language like Java. There are also several DSLs implemented on MapReduce, including Sawzall [19], Cascading [6], and Papyrus [4]. These DSLs facilitate parallelism by providing building blocks with which the application programmer can construct parallel dataflows, but they have two shortcomings. First, they may not be suitable for all scientific workloads because they are designed for data-intensive workloads. Second, they require a modest understanding of parallel programming even though they hide many of the details of distributed computing. Through the development of MapReduce-backed specializers, our work enables scientists to leverage the power of these languages without the need to design MapReduce workflows.

Liszt [14] is an example of a DSL embedded in Scala that leverages inter-machine parallelism to solve mesh-based partial differential equations. The Liszt compiler uses domain-specific knowledge to uncover parallelism and to generate a platform-independent intermediate representation that can be compiled for SMPs, GPUs, and clusters via MPI. The performance of Liszt programs is competitive with low-level, hand-tuned implementations. The success of Liszt highlights the need for approaches like SEJITS that enable rapid development of high-performance, platform-portable DSLs. Indeed, Liszt could be implemented in SEJITS by a specializer that uses domain-specific analyses to target the existing MPI, CUDA, and Cilk backends.

Scientific computing on the cloud.

The cloud is becoming an increasingly popular platform for scientific computing. Several projects like ours provide a path to the cloud for scientists. Pegasus [13] is a system for managing scientific workflows. It supports cloud-based Condor clusters, but applications must be casted as Pegasus workflows, the user must have access to a local Condor central manager node, and cloud nodes must be accessible from machines outside the cloud. In contrast, our solution frees the scientist from this configuration by embedding it in the specializer.

Neptune [9] is a domain-specific language that allows scientists to deploy and utilize cloud-based HPC clusters. It supports applications built on MPI, X10, UPC, MapReduce, and others, and provides conditionals for controlling cloud resource consumption based on application results. Scientists using Neptune must learn its Ruby-like syntax, and they must explicitly write applications in the supported parallel computing paradigms. Our approach requires knowledge of no additional languages, and only the expert specializer developer must interact with the parallel framework.

StarCluster [5] is utility for managing HPC clusters on Amazon's Elastic Compute Cloud. Scientists can launch clusters using custom virtual machines or ones pre-configured with Hadoop, MPI, and scientific libraries via a command-line interface. StarCluster provides no support for controlling jobs on the cluster, and scientists must explicitly write applications for the supported parallel frameworks.

3. SYSTEM ARCHITECTURE

In this section we describe our SEJITS implementation and the extensions required to target clusters via MapReduce. Our primary implementation is called Ass (A SEJITS for Python) and as the name suggests, it is implemented in Python. Asp provides a Module object as the interface between specialization logic and the collection of backends. Specializers are responsible for registering backend-specific specialization methods with the Module, which then interposes in application logic to apply them. Modules use Python decorators to intercept calls to specialized entities such as classes, methods, and functions. Decorators supply the entities' abstract syntax trees to specialization logic, so transformations can be arbitrarily complex.

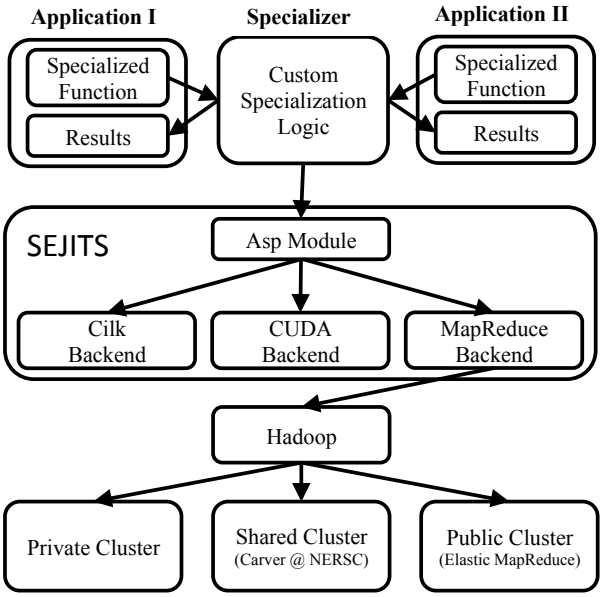


Figure 1: The position of the MapReduce backend in the SEJITS architecture.

```
class AspMRJob(mrjob.MRJob):

  def mapper(self, key, value):
    """
    Yields 0 or more key/value tuples.
    """

  def reducer(self, key, values):
    """
    Yields 0 or more key/value tuples.
    """

  def cache_task_inv_data(self, data):
    """
    Stores task-invariant data in Hadoop
    Filesystem (HDFS).
    """

  def retrieve_task_inv_data(self):
    """
    Retrieves task-invariant data from HDFS.
    """
```

Figure 2: The simplified interface to the MapReduce backend. Because the backend is built on MRJob, it also supports jobs without reduce steps and multi-step jobs.

Our implementation can be logically divided into three components: a new MapReduce backend, infrastructure for caching task-independent data, and a connection to Asp's application tuning system.

3.1 The MapReduce Backend

The MapReduce backend presents an interface through which the specializer developer can run MapReduce jobs. Figure 1 shows the position of the MapReduce backend within the SEJITS architecture, and Figure 2 shows the interface provided to the specializer developer by the backend. To manage interaction with Hadoop, the backend is built on MRJob [3], an open-source package for writing MapReduce programs in Python. MRJob provides support for both private clusters and Amazon Elastic MapReduce. It utilizes a feature of Hadoop called Hadoop Streaming, wherein key-value records are passed between mapper and reducer executables via pipes. This incurs additional overhead due to object serialization and inter-process communication, but it allows mapper and reducer executables to be implemented in any language.

The interface to the MapReduce backend is built on MRJob's interface and includes some additional features specific to specialization. Specializer developers construct MapReduce jobs by subclassing the AspMRJob class, shown in Figure 2. As in traditional MapReduce, jobs are broken down into map and reduce operations. Reduce operations can be omitted, and multi-step, "iterative" operations can be defined in which output records from reducers are fed into the next round of mappers. Although not shown in Figure 2, MRJob also supports iterative jobs.

Once a specializer defines an AspMRJob class, it must register it with Asp. From this point, the specialized application behaves as follows:

1. The application executes the definition of a specialized class, method, or function. In normal Python, this inserts a named reference to a callable object into the current namespace. Instead, Asp interposes and presents the object's abstract syntax tree (AST) to the specializer.
2. The specializer performs arbitrary, developer-specified analysis of the AST.
3. Based on the analysis, the specializer returns a callable object that will serialize its arguments, execute the MapReduce job, de-serialize the result, and return it.
4. The application calls the callable object. This actually executes the MapReduce operation. When the operation terminates, the call returns as if it had executed normally.

This architecture enables the exploitation of inter-machine parallelism in a manner that is invisible to the application programmer.

3.2 Data Caching Framework

Our experience implementing two MapReduce-backed specializers (described in the next sections) prompted us to add functionality for caching task-invariant data. We first implemented "functional" mappers: output records were solely a function of input records. Unfortunately, our problem required 16MB for each of 200,000+ records, so this approach required infeasible amounts of memory. We observed that all but 276 bytes of each input record were invariant across map tasks, so we implemented a facility for caching task invariant data in Hadoop Filesystem (HDFS). Our mechanism serializes input data, uses an MRJob feature to distribute it to the cluster, and provides a convenient method to access it from each mapper and reducer. The specializer developer is responsible for identifying task invariant data, and caching is invisible to the application programmer.

3.3 Asp Tuning Integration

Specialization occurs at runtime, so characteristics of both the target platform and the input data can inform optimization logic. Asp provides support for defining abstract an tuning space and selecting from it the best known parameters for a particular problem. However, finding the best tuning parameters is complicated by the non-uniformity of the tuning space. Therefore, Asp exhaustively searches the space and stores the results in a global, institution, or local database. We hope that better performance models or machine learning techniques will provide a better way to identify the optimal tuning parameters.

There exist opportunities for tuning in the context of the MapReduce backend. Hadoop clusters provide an array of configurable parameters that specializers can adjust for better performance. Notable parameters include the number of reduce tasks, the data replication factor of HDFS files, and the number of records per map task. The latter was tuned to yield a 3x speedup on our first specializer, as described in the next section. Tuning clusters in the cloud is further complicated by increased variability in performance due to virtualization. Our implementation does not attempt to solve the tuning problem, instead it uses Asp's built-in exhaustive search to find the best parameters..

3.4 Limitations

Our system requires that the environment in which MapReduce tasks execute mirrors the application developer's environment. More specifically, the Python interpreter versions must match and any libraries referenced by the specializer must be available on the cluster. Specializer developers can denote source files to be transferred to the cluster dynamically, but this approach will not suffice for all applications.[1] Our implementation does not attempt to solve the "same-environment" problem.

In our experience, the "same-environment" requirement is easy to satisfy on persistent clusters with global file systems. It is more difficult on Amazon Elastic MapReduce's shared-nothing instances because EMR boots vanilla instances for each job. We developed scripts to install all software dependencies for each shared-nothing instance at boot. The development cost of setting up the remote environment also depends on the application: specializers with substantial software dependencies may require a significant investment to build their environments, whereas some applications may only require the default Python installation.

4. CASE STUDIES

This section presents case studies for two applications using MapReduce-backed specializers. Each study includes a description of the specializer design, performance results, and a discussion of the efficacy of our approach.

4.1 FTDock Case Study

FTDock [7] is a Python application that finds the best fit between two protein molecules in a six-dimensional rotational and translational space. It accelerates the translational comparison by convolving[2] the discretized molecules, so for our purposes the application performs two nested searches: one of the 3D rotational space, and another of the 3D translational space for each rotational configuration. FTDock scores the fit for each configuration and returns the best. To find optimal fits, the search must be exhaustive because the space of scores is highly irregular. Fortunately, fits are independent so they can be performed in parallel.

[1]For example, a mapper might call a function in a custom Python library, implemented in C. The function must be available on the cluster, so a specializer could copy the binary to the cluster. Obviously, this would be pointless unless the systems were binary-compatible and all other dependencies were met.

```
1 class MapMRJob(AspMRJob):
2   def mapper(self, fxn, args):
3     iargs = self.retrieve_task_inv_data()
4     result = fxn(*args, *iargs)
5     yield 1, result
```

Figure 3: The MapReduce class for the FTDock specializer. Additional code for cluster tuning and access credentials have been omitted for clarity and privacy.

4.1.1 FTDock specializer

We developed a MapReduce-backed specializer to bring massive parallelism to FTDock. The specializer interposes in application logic at the point where the scoring function is applied to each point in the rotational search space. There, the specializer serializes the scoring function[3], the set of points in the search space, and the protein molecule objects. It constructs a set of MapReduce input records of the form:

```
(key, value) = (fxn, (point, proteins))
```

The specializer then launches a MapReduce job to process the set of records. The FTDock specializer manages its MapReduce workflow with the MapMRJob class, shown in Figure 3. The MapMRJob class applies a given function, the record's key, to every datum, the record's value, and returns the results. It was simple to cast FTDock's 3D search in this form: the scoring function is the key, and the value is a tuple of the molecules and the rotational configuration in question. A reduction step is unnecessary, so output records are emitted with the same key for easy post-processing.

Upon completion of the MapReduce job, the specializer parses the results by stripping the key from the set of values. It returns these values, the scores for each orientation, to the application.

4.1.2 Specializer Optimizations

We made one optimization to reduce the memory space consumed by the set of input records. Because serialized protein objects are large in size and invariant across all fittings, we store them in HDFS via AspMRJob's `cache_task_inv_data` method. Thus, input records to MapMRJob job become:

```
(key, value) = (fxn, point)
```

For standard problems with hundreds of thousands of input

[2]Convolution is implemented by multiplications in Fourier space, hence FTDock.

[3]Serialization is implemented with Python's `pickle` module, which serializes the function's fully-qualified name. During de-serialization, the name must be importable; this is enforced by the "same-environment" requirement described in Section 3.4.

records, this reduced memory requirements from terabytes to tens-of-megabytes.

4.1.3 Experimental Conditions

To measure the performance of the FTDock specializer, we conducted experiments on two Hadoop platforms: Carver, a shared cluster, and a cloud-based cluster in Amazon Elastic MapReduce. Carver is a 75-node cluster at the National Energy Research Supercomputing Center (NERSC). Each Carver node has two quad-core Intel Xeon processors (2.67 GHz, Nehalem). Carver's Hadoop installation dedicates 450 cores to map tasks and 150 to reduce tasks. On Amazon, we ran on a cluster of 10 Cluster Compute Instance, each with two quad-core Intel Xeon processors (2.93GHz, Nehalem). The Amazon cluster had map- and reduce-task capacities of 120 and 30, respectively. Finally, we ran a serial benchmark of the application on a 3.2GHz Core i7 processor to estimate the best single-machine performance.

Throughout the experiments, we tested the same pair of protein molecules, identified by 2ST1 and 2CI2 in the World-wide Protein Databank [8]. All tests were performance at the molecular grid resolution of 0.75 Å, the preferred size of our application users.

4.1.4 Experimental Results

To compare the performance of different platforms, we used an application level metric: the throughput of each platform in dockings per second. Figure 4 shows the performance of all platforms as a function of the problem size. Each curve has two domains: a rising edge for loads within the immediate capacity of the platform, and a plateau indicating a saturation of the platform's capacity. The three curves representing serial runs level off quickly, indicating that throughput is saturated even for small problem sizes. Throughputs of all three serial runs differ by a small constant, suggesting that faster processors have little effect on throughput. In contrast, the specialized runs achieve much higher throughput. The 120-core Amazon cluster yields a 52x speedup over serial for 50,653 fits, the largest problem size tested. Similarly, the 450-core Carver yields a speedup of 264x over serial for problems with 389,017 fits. From the scientists' perspective, a 264x speedup reduces total execution time from 13 days to 73 minutes. Finally, weak performance for small problem sizes is irrelevant because scientists are only interested in problems with hundreds of thousands of fits.

Figure 5 show the percentage of Hadoop overhead for all problem sizes, calculated as a fraction of the serial execution time t_s, the cluster capacity C in cores, and the parallel execution time t_p:

$$\%_{overhead} = 1 - \frac{t_s}{C \cdot t_p}$$

As expected, overhead per fit is amortized across all fits for large problem sizes. Even so, overhead is significant for large problems. This suggests better tuning and alternate frameworks for distributed computing may yield additional speedups.

4.1.5 Tuning Results

We investigated the effect of tuning two parameters: the number of dockings per map task and the HDFS data replication factor.

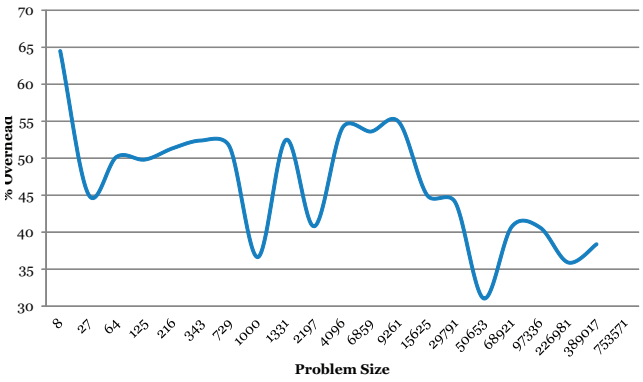
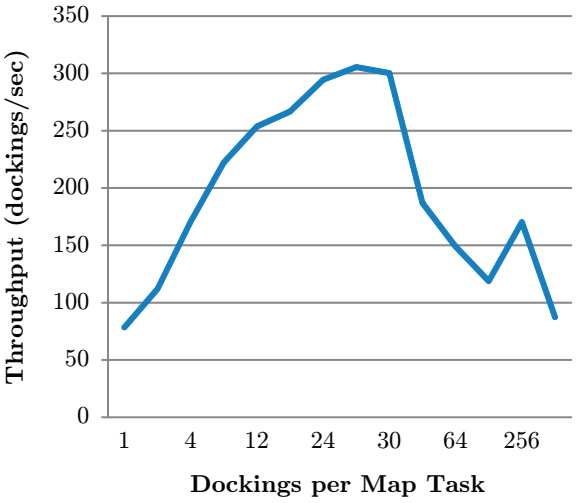
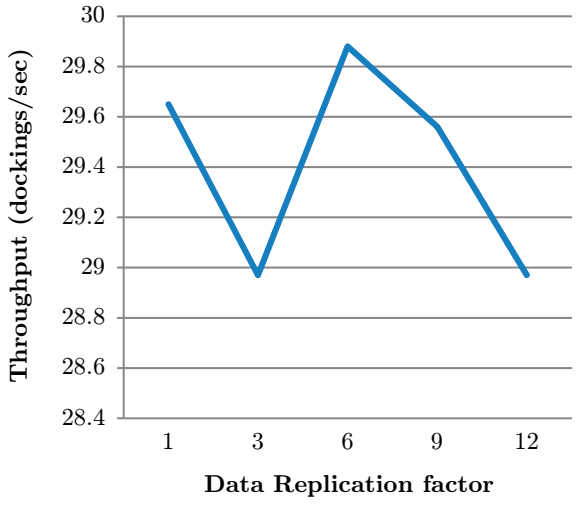


Figure 5: Hadoop's overhead for FTDock



(a) Throughput vs. Docking Per Map Task



(b) Throughput vs. Replication Factor

Figure 6: Tuning results for two parameters.

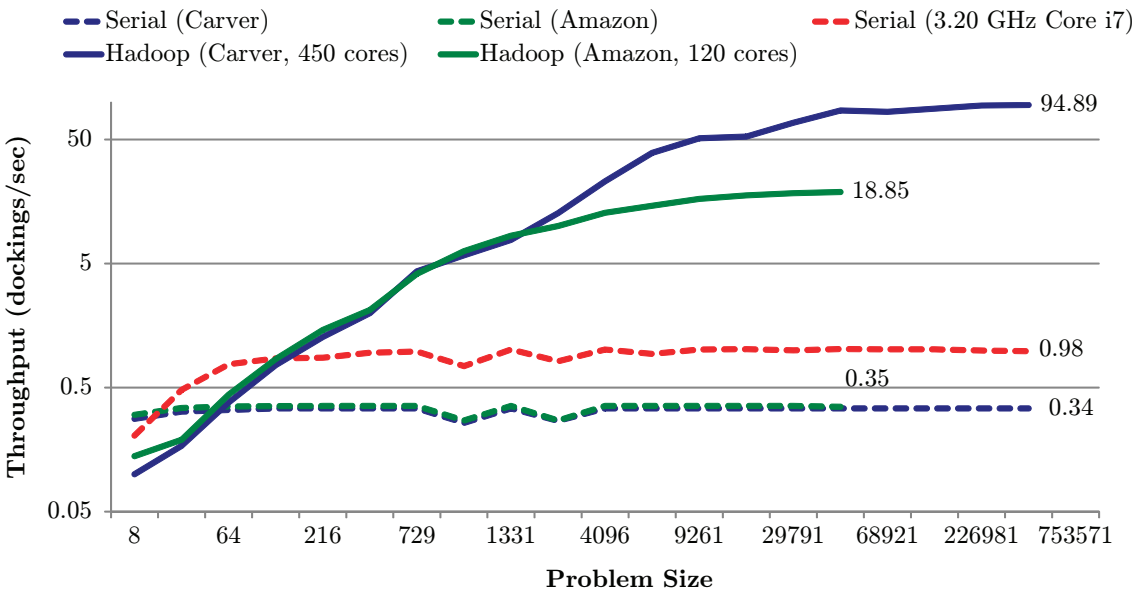


Figure 4: Performance results for FTDock. Scientists require 100,000+ problem sizes for acceptable resolution.

Dockings Per Map Task.

Hadoop allows users to specify the number of records (dockings) per map task. This parameter essentially defines the unit of work in Hadoop: excessively small units incur additional overhead per record, and excessively large units of work create load imbalance. This phenomenon is shown in Figure 6, which shows the throughput as a function of the number of dockings per map task. The correct choice of value yields a 3.9x speedup over the worst choice, and thereby demonstrates the importance of tuning.

Data Replication Factor.

The data replication factor controls the number of replicas in HDFS for a given file. We expected to improve read bandwidth of the protein data file by increasing the number of replicas, but the effect was minimal. Figure 6 shows the throughput as a function of the data replication factor. As the graph suggests, the effect of wide replication is not statistically significant. Such a result indicates that the FTDock specializer is not limited by read bandwidth.

4.1.6 *Strong Scaling of the FTDock Specializer*

Comparisons of throughput for both the 120-core Amazon cluster and the 450-core Carver cluster can provide a limited indication of the strong-scaling properties of the FTDock specializer. Serial executions on both Carver and Amazon exhibit roughly the same throughput, so one would expect that cluster throughputs are proportional to the number of cores in the cluster. For the 50,653 problem size, the Carver cluster ran 3.86x faster that the Amazon cluster, and the Carver cluster had 3.75x more cores. We believe the specializer fell short of perfect strong scaling because of higher variability in the cloud environment, whereas Carver has a dedicated rack for Hadoop.

The ability to run on the cloud allows scientists to scale their jobs to the optimal execution time and monetary cost point. To demonstrate this capability, we performed a complete docking in under one hour on Amazon's cloud. We ran on 720 cores for 57 minutes, incurring $97.44 in usage fees. We expect the ability to scale will valuable to scientists with impending deadlines and large budgets.

4.1.7 *Discussion*

Our results indicate that specialization via MapReduce is an attractive means of accelerating FTDock. With a specializer and a one-line change to an application, the domain scientist can realize hundred-fold speedups on private clusters or the cloud. Furthermore, the scientist can scale her cloud-based clusters to the optimal time/cost point to conserve funding or meet deadlines. Finally, our results indicate that casting programs as MapReduce workflows is a viable technique for extracting inter-machine parallelism in real-world applications like FTDock.

4.2 Speaker Diarizer Case Study

The previous section has demonstrated our framework's ability to achieve good performance and scalability for a pure Python program. In this section, we demonstrate its ability to compose parallel programming patterns within our second benchmark program, Speaker Diarizer.

4.2.1 *Speaker Diarization*

Speaker Diarizer is an application that determines 'who said what' during a meeting, as captured in an audio file. The diarization process consists of several steps. First, the program performs a uniform segmentation of the input audio file into multiple segments much greater than the expected number of speakers. Then, it constructs Gaussian Mixture Models (GMMs) representing features of each audio segment. Next, the program iteratively re-segments the

audio segments based on their GMMs, re-trains the GMMs, and permanently merges two GMMs with most likelihood together. These steps are repeated until no more GMMs can be merged.

Determining two most similar GMMs involves scoring all possible pairs of GMMs based on Bayesian Information Criterion (BIC), a computationally-intensive task. Our benchmark offers two choices of specializers for accelerating parts of the BIC score calculation: a Cilk backend for multi-core CPUs and CUDA backend for GPUs.

4.2.2 *Speaker Diarizer Specializer*

The CUDA- and Cilk-targeting specializer does not take advantage of the embarrassingly-parallel nature of BIC score calculations. Therefore, we hoped to achieve a speedup by running BIC calculations in parallel with MapReduce. We wrap the original BIC calculation with the map function in our MapReduce-backed specializer.

During execution, the specializer intercepts a list of GMMs and determines if a MapReduce platform is available. If not, it continues with the original code segment (which processes GMM pairs serially). If a MapReduce platform is present, it generates all possible pairs of the GMMs in the list and launches a MapReduce job to process them. The job distributes all GMM pairs to the mappers, and each mapper computes the BIC score on local GPUs by calling the existing, GPU-backed specializer. Mappers then pass the pair and its score to the reducers, which simply compare all scores and return the best pair of GMMs to the application.

4.2.3 *Test Environment*

Our tests were conducted using Amazon's Elastic MapReduce (EMR) service on Cluster GPU (cg1.4xlarge) instances. Each instance provides 2 Intel Xeon x5570 quad-core processors, 2 NVIDIA Tesla M2050 GPUs, 22GB of memory, 1690GB local storage, and 10Gbps Ethernet. An instance can process up to 12 concurrent map tasks and 3 concurrent reduce tasks. All tests were run in a CUDA 4.0 environment on 64-bit Linux operating system.

We used the 30-minute meeting audio set "1006b" from the Augmented Multi-Party Interaction (AMI) corpus as our test set. For most tests, we configured the number of initial audio segments to 16, since it is the typical number of segments for a 30-minute test set [16].

4.2.4 *Performance Results*

We first investigated the strong scaling properties of the specializer. Figure 7 shows the execution time as a function of the number of core dedicated to map tasks. The results reveal that using MapReduce actually hinders performance because overhead dominates the actual computation. We have identified three factors responsible for this underwhelming performance:

Network Bottlenecks. All pairs of GMMs, which total 700MB for this application, must be distributed across mapper node at the beginning of each iteration. In addition, the test implementation passed the serialized GMMs to reducers so that they could return the optimal GMM pair. This is sub-optimal and has been replaced by passing references in more recently implementations.

Multi-stage Mapping. The all-pair BIC score computation is invoked in every iteration because the specializer can only interpose at certain points in the application. With small changes to the application, the specializer could identify the parallelizable BIC operation inside a loop and construct a corresponding, iterative MapReduce operation. An iterative MapReduce operation would obviate the need for transferring all GMM pairs to and from HDFS, and it would only incur the startup costs once.

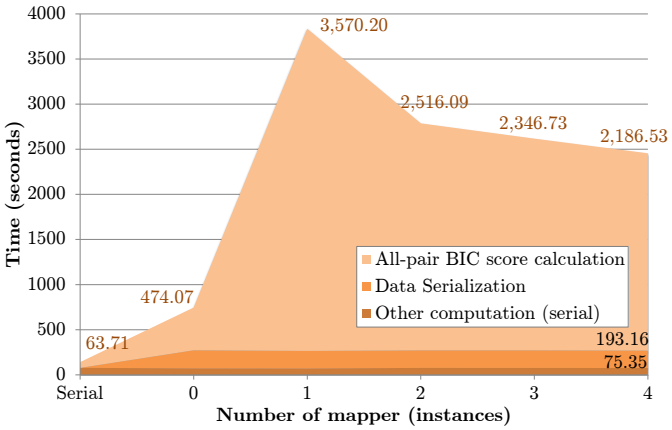


Figure 7: Runtime breakdown for each number of mapper instances (1006b test set, 1 GMM pair per map task)

Hadoop Scheduling. Figure 8 shows the total overhead time as a function of the problem size for various cluster sizes. The overhead for the last two iterations is significantly larger than previous ones due to its much smaller problem size. In addition, in nearly half of all experiments, the MapReduce jobs failed during the last two iterations, causing the application to terminate. We believe this is due to resource contention in the virtualized environment.

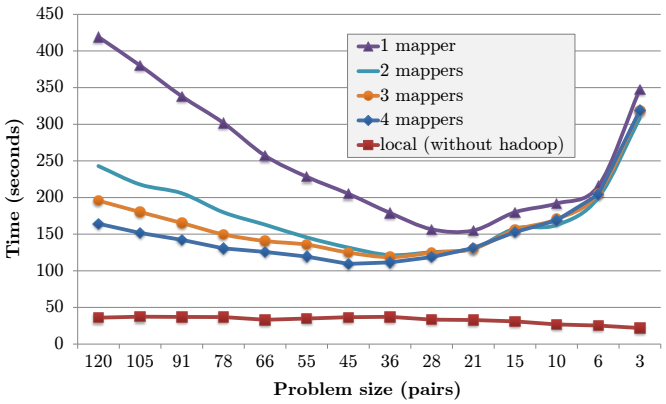


Figure 8: Hadoop overhead during all-pairs BIC calculation for each iteration (1006b test set, 1 pair per map task)

The time required for data serialization is also nontrivial. Our current implementation uses Python's `pickle` mod-

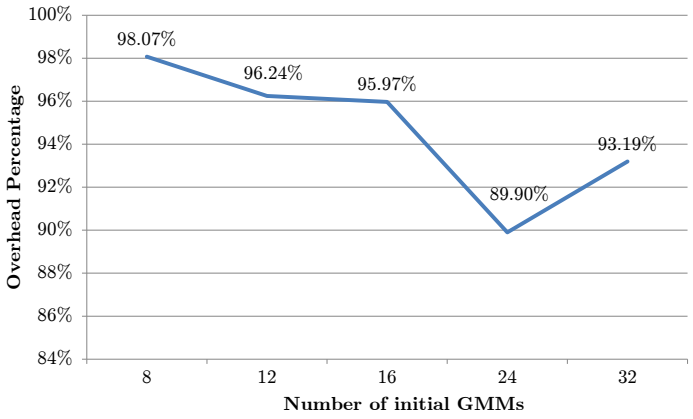


Figure 9: Overhead percentage per BIC calculation as a function of initial GMMs segments (1006b test set, 3 map task workers, 8 pairs per map task)

ule to serialize data, which takes approximately 193 seconds for the IS1006b test set. Even though there is some room for improvement, the application is already heavily optimized. The CUDA-backed specializer provides 3x performance, bringing the total execution time within 63 seconds. These factors makes it impossible for the MapReduce-backed specializer to achieve better performance for the given test set.

We also evaluated the program on various number of initial audio segments, since it determines the number of iterations and hence the problem size. Figure 9 shows the average overhead percentage per one pair calculation of different initial audio segments on 3 mapper instances. The overhead for the run with 24 initial GMMs case is lower than the run with 32 because the application was aborted before completing its last few iterations whose overheads are usually much higher than others. Even though the overhead is amortized as the number of initial GMMs increases, it is still unlikely to match the serial version, as the input size increases as well. Moreover, it is also unlikely that there will be any run of number of initial segments greater than 32 in practice, since 16 is the typical number.

4.2.5 *Optimizations*

Due to our limited EMR budget, we only explored two optimizations.

GMM Pass-By-Reference Our original implementation passed serialized GMM pairs to reducers so the best could be immediately returned to the application. This resulted in wasted cycles, storage space, and network bandwidth, so we updated the mapper function to emit scores tagged with GMM identifiers. When the reduce phase yielded the best pair of identifiers, the specializer could retrieve the corresponding GMMs and return them to the application.

Number of Pairs Per Map Task Finding the best number of pairs per map task was more complicated than in FTDock. FTDock only launches one MapReduce operation per run, so it trivially has the same problem size throughout its execution. In contrast, Speaker Diarizer launches numerous MapReduce jobs, each with different problem sizes. Therefore, the best number of pairs per map may differ over the course of execution. We determined that the static value of 8 provided reasonable performance, and we expect that the tuning system will provide the best approach for dynamically determining the optimal number of pairs per map task for given problem sizes.

4.2.6 *Discussion*

Despite the substantial performance degradation, Speaker Diarizer demonstrates our solution's ability to compose parallel programming patterns. By composing the MapReduce-backed specializer with the existing GPU-backed specializer, we were able to exploit both intra- and inter-machine parallelism. The performance results suggest that our framework is not suitable for applications with only a moderate amount of exploitable parallelism. To amortize the overhead of MapReduce, the application must include many independent, compute-intensive tasks. Finally, it suggests that our framework should perform a cost analysis to determine whether use of the cloud will be profitable.

5. FUTURE WORK

We will pursue future work along two dimensions. First, to minimize the overhead associated with distributed computing, we plan to augment our backend to target other MapReduce-like platforms. Spark [21] accelerates iterative MapReduce jobs by keeping the working dataset resident in memory, and Clustera [15] and Dryad [17] are designed to manage both I/O- and computationally-intensive workloads. Even though these platforms are not as widely deployed as Hadoop, they may provide better strong scaling performance.

Second, we plan to develop cluster-targeting backends for applications that cannot be cast as efficient MapReduce workflows. We expect that parallel programming abstractions in more general distributing computing frameworks will provide valuable building blocks for specializer developers writing tightly-coupled, synchronous applications.

6. CONCLUSION

This paper has described a framework for exploiting inter-machine parallelism by enabling expert programmers to cast application logic as MapReduce jobs and execute them on clusters of machines. MapReduce-enabled specializers allow data-parallel applications to scale efficiently to clusters with hundreds of cores and in doing so, establish clusters as viable targets for specialization. Specializers targetting Amazon Elastic MapReduce are even more useful because the illusion of infinite resources allows scientists to scale simulations to the optimal duration-cost point. Finally, this work demonstrates a technique for achieving multi-level parallelism by composing single-machine specializers within MapReduce workflows. We believe this composition of parallel programming patterns will prove useful as core counts increase and systems become more heterogeneous.

7. REFERENCES

[1] http://aws.amazon.com/elasticmapreduce/.
[2] http://hadoop.apache.org/.
[3] http://packages.python.org/mrjob.
[4] http://rubygems.org/gems/hadoop-papyrus.

[5] http://web.mit.edu/stardev/cluster.

[6] http://www.cascading.org.

[7] http://www.sbg.bio.ic.ac.uk/docking/ftdock.html.

[8] http://www.wwpdb.org/.

[9] C. Bunch, N. Chohan, C. Krintz, and K. Shams. Neptune: a domain specific language for deploying hpc software on cloud platforms. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, ScienceCloud '11, pages 59–68, New York, NY, USA, 2011. ACM.

[10] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. Technical report, EECS Department, University of California, Berkeley, Sep 2010.

[11] B. Catanzaro, S. A. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. A. Yelick, and A. Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, Mar 2010.

[12] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. *Data Parallel Haskell: a status report*, pages 10–18. Number Damp. ACM, 2007.

[13] E. Deelman, G. Singh, M. hui Su, J. Blythe, A. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13:219–237, 2005.

[14] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[15] D. J. DeWitt, E. Paulson, E. Robinson, J. Naughton, J. Royalty, S. Shankar, and A. Krioukov. Clustera: an integrated computation and data management system. *Proc. VLDB Endow.*, 1:28–41, August 2008.

[16] D. Imseng and G. Friedland. Robust speaker diarization for short speech recordings. In *Proceedings of the IEEE workshop on Automatic Speech Recognition and Understanding*, number Idiap-RR-26-2009, pages 432–437, PO Box 592 CH - 1920 Martigny, 12 2009.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41:59–72, March 2007.

[18] H. Pan, B. Hindman, and K. Asanovic. Composing parallel software efficiently with lithe. In B. G. Zorn and A. Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 376–387. ACM, 2010.

[19] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13:277–298, October 2005.

[20] L. Ramakrishnan, P. T. Zbiegel, S. Campbell, R. Bradshaw, R. S. Canon, S. Coghlan, I. Sakrejda, N. Desai, T. Declerck, and A. Liu. Magellan: experiences from a science cloud. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, ScienceCloud '11, pages 49–58, New York, NY, USA, 2011. ACM.

[21] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.