Scalable Distributed Image Transcoding using Python-WorkQueue

Jeffrey Westphal and Peter Bui
Department of Computer Science
University of Wisconsin - Eau Claire
Eau Claire, WI 54702
{westphjm, buipj}@uwec.edu

Abstract

Transcoding large amounts of digital media from one format to another is a common data intensive workflow. In this paper, we present a scalable image transcoding system based on Python-WorkQueue that significantly reduces the amount of time required to convert images from one format to another by mapping transcoding tasks across a distributed pool of remote workers. We test our system using a Condor cluster and varying amounts of files and number of workers. Our results show that we are able to achieve speed increases up until a certain limit.

1 Introduction

Given the rapidly growing deluge of data in both scientific research and industrial applications, there is an increasing need for scalable and effective solutions for processing large amounts of data in a timely manner [6]. A common data intensive workflow that consumes a great deal of time is the transcoding or conversion of media files from one format to another. For instance, videos uploaded to YouTube are transcoded into a variety of formats and resolutions. Likewise, data in a scientific experiment is usually collected in a raw format that is later transformed into a more usable format. To speedup the process of transcoding a large number of files, distributed systems are often used to divide the work across multiple machines. Unfortunately, programming applications that work in such environments can be difficult and complex.

One common approach to processing large amounts of data is to use the MapReduce [4] abstraction. In this programming model, developers only need to specify **(1)** a *map* function that filters, selects, or transforms an input dataset and **(2)** a *reduce* function that aggregates or collects the results. The major advantage of this framework is that the complexities of distributed programming are abstracted away from the programmer, who is then free to focus on the particular domain application rather than low-level systems details. Today, many companies and researchers utilize Hadoop [5], an open source Java implementation of MapReduce, to perform extensive data analysis on large datasets.

Although MapReduce is an effective and powerful data processing abstraction, it is unnecessary for transcoding media files from one format to another. For this type of workflow, we simply need a *Map* abstraction as shown in Figure 1.

```
1 def Map(function, dataset):
2     results = []
3     for data in dataset:
4         results.append(function(data))
5     return results
```

Figure 1: Map Abstraction

The *Map* abstraction involves applying a user-specified *function* to every item in an input *dataset* in order to produce another dataset. Because each element can be processed independently, this pattern of computation is considered *naturally parallel* and thus can be performed concurrently. For a transcoding workflow, the *function* is usually a conversion application such as ImageMagick [1] and the *dataset* is the collection of media files.

In this paper, we present our prototype of a distributed *Map* abstraction implementation based on the Python-WorkQueue [3] framework. We evaluate this system by utilizing it to perform a series of image conversions on different datasets. Our results show that the image transcoding system significantly reduces the running time of the conversion workflow by splitting the transcoding of files across a distributed computer cluster. At the end of the paper, we discuss how much speedup is gained and analyze the limitations to our approach.

2 Design

In order to create a scalable and distributed *Map* abstraction, we utilize Python-WorkQueue to coordinate the computation in our workflow. Python-WorkQueue is a Python binding to the WorkQueue master-worker distributed computing framework. In the master-worker paradigm, a *master* application creates a set of *tasks* and sends it to a pool of *workers* who perform the specified computation and return the results. Because the workers can execute on different machines, WorkQueue applications are *distributed*. Likewise, WorkQueue applications are *scalable* because the framework allows for workers to come and go, and handles fault-tolerance for the developer. Using Python-WorkQueue, our image transcoding system internally implements the *Map* pattern to construct a distributed workflow that is executed by a pool of workers running on multiple machines.

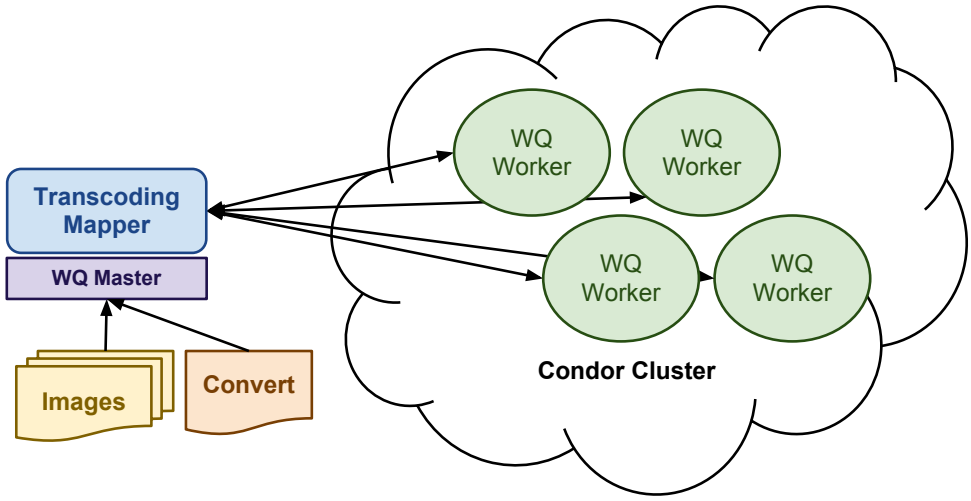

Figure 2: Transcoding System Overview.

The overall design of our image transcoding system is shown in Figure 2. Using the master-worker paradigm provided by Python-WorkQueue, our application implements the *Map* pattern to schedule one conversion task per image in the input dataset as follows:

1. For each image in the input data set, our application creates a transcoding task consisting of the input image, the conversion *function*, and the parameters for the output image. In our workflow, the transcoding *function* is the ubiquitous ImageMagick `convert` utility.

   To ensure that `convert` works on any remote worker, we utilize the `starch` [9] application archiver to produce a *standalone application archive* capable of executing on different machines even if they lack the necessary libraries.

2. Once these tasks are scheduled, the application waits as the internal WorkQueue master distributes the tasks to a pool of remote workers. In our setup, these workers execute on a local Condor pool and contact the master for tasks to execute. When contacted by the workers, the master sends the transcoding tasks to the workers by

transferring an input image file and the transcoding executable to the remote machine who then performs the appropriate task.

3. Eventually, all of the tasks are completed and the results of the computations are returned to the transcoding application. If a worker is evicted or crashes, the WorkQueue library will internally re-schedule that task to run on another worker and thus handles failure transparently for the user. If additional workers join the pool, the master will utilize these as long as there are tasks to be execute and thus the application can scale dynamically at run-time.

By utilizing the *Map* abstraction in conjunction with Python-WorkQueue, we are able to implement a scalable distributed image transcoding system.

3 Methodology

This section provides further details about our implementation of a scalable distributed image transcoding system and how we evaluated its effectiveness.

As noted in the previous section, we utilize Python-WorkQueue to manage the complexities of coordinating a distributed pool of remote workers. The bulk of transcoding implementation therefore involves utilizing the *Map* pattern to schedule the appropriate conversion tasks and waiting for them to execute.

```
# Path to files to be converted
path = sys.argv[1]
files = os.listdir(path)

# For each file in path, determine if it is an image and then schedule it
for filename in files:
  infile = os.path.join(path, filename)

  # Check if file is an image
  if imghdr.what(infile) is not None:
    outfile = os.path.splitext(filename)[0] + '.' + sys.argv[2]
    command = './convert.sfx %s %s' % (filename, outfile)

    # Create WorkQueue Task
    t = Task(command)

    # Specify input and output files
    t.specify_file('convert.sfx', 'convert.sfx', WORK_QUEUE_INPUT)
    t.specify_file(infile, filename, WORK_QUEUE_INPUT)
    t.specify_file(outfile, outfile, WORK_QUEUE_OUTPUT)

    # Submit Task to Master
    q.submit(t)
```

Figure 3: Task Construction Code.

Figure 3 contains the portion of our application that handles the scheduling of the tasks. The process of generating tasks goes as follows:

1. After instantiating the WorkQueue master object, our application grabs the list of files to process from the command line.

2. Next, the program loops through all files found and generates a complete path to each file and determines whether a given file is, in fact, an image. If it is not, the file is passed over and the next file is tested.

3. If the file is an image, our application must then create a WorkQueue task object consisting of the command to run, the input files to send, and the output file to receive.

   As noted previously, we utilized a *starched* version of the ImageMagick convert command as the transcoding function. Because our system is operating in a heterogeneous distributed environment, we have to send both the data and the executable to the remote worker. Fortunately, by specifying in the task object, WorkQueue will manage the data for us and will even cache files between tasks. This means that the `convert.sfx` is only transferred to the remote worker once.

4. Once the task is fully specified, we then submit it to the WorkQueue master object who will send it to remote workers when they become available.

As can be seen, the process of scheduling tasks is similar to the pattern in the *Map* abstraction. Rather than applying the function immediately, our transcoding system creates a task which serves as a *future* or *promise* [2]. That is, the actual execution of the function is delayed until a later time. To retrieve the results of these delayed computations, the tasks are then executed concurrently across a pool of remote workers, which enables an increase in the throughput of the entire transcoding workflow.

```
1 # As long as the queue is not empty, wait for a task to complete
2 while not q.empty():
3    t = q.wait(5)
4    if t:
5      print "task (id# %d) complete: %s (return code %d)" % \
6        (t.id, t.command, t.return_status)
7      print t.output
```

Figure 4: Task Waiting Code.

Figure 4 shows how our transcoding system waits for all the tasks to complete. To do this, we simply loop until the WorkQueue master's task queue is empty. As long as there are tasks in the queue, we wait and simply print out the results of a completed tasks. Because WorkQueue manages fault-tolerance and data transfers for us, we do not need to manually reschedule tasks from failed workers or retrieve output files. Instead, we simply wait for the tasks to complete. In total, our transcoding system consisted of `82` lines of Python code.

We benchmarked our solution to test for scalability and performance by running a series of image transcoding workflows on our local Condor cluster. This system consists of thirty CPU cores on five machines (some physical and some virtual). Each computer in the cluster runs the 64-bit version of CentOS 6 and shares a repository of software via NFS.

```
#!/bin/bash

# Run our distributed convert program
# $1 Number of Workers
# $2 Number of Files
# $3 Image Files
benchmarking() {
  ./work_queue_convert $3/$2 png $1 $2
  rm *.png
}

# Loop over sets of images
for t in archlogo galaxy falls
do
  # Loop over numbers of workers
  for i in 1 2 4 8 16 24 30
  do
    # Create $i number of workers on the Condor cluster
    work_queue_pool -T condor dplsubmit.cs.uwec.edu 9123 $i &
    id=$!
    # Loop over numbers of files that a trial will run
    for j in 10 100 1000
    do
      # Loop to run individual trials
      for k in {1..10}
      do
        benchmarking $i $j $t
      done
    done
    # Terminate work_queue_pool and all workers it is maintaining
    kill $id
    sleep 5
  done
done
```

Figure 5: Benchmarking Shell Script.

To ensure consistency in our experiments, all source files were JPEG images and were converted to PNG format. Three image sizes are used: fifteen kilobytes, one megabyte, and ten megabytes. Additionally we organized the images into sets of ten, one hundred, and one thousand files of each size. For each of these combinations of group size and image size, we utilized our application to transcode the images and timed the execution to determine the length of time necessary to convert each set of images using a given number of WorkQueue workers. We varied the number of workers in the following increments: 1, 2, 4, 8, 16, 24,

and 30. To further ensure consistency of our results, we measured the execution time of the transcoding application across multiple trials for each combination of image size, group size, and worker count.

In order to test our application in an automated manner, we use the shell script shown in Figure 5. The script first loops over the images and number of workers that will be used. The script uses `work_queue_pool`, a utility designed to create, destroy, and maintain WorkQueue workers to ensure the proper number of workers exist at a given time. The script starts the pool in the background and saves its process id. The script then loops over the number of images and then runs ten trials for each combination of image, number of workers, and number of files by passing this information to a function that runs our application, referred to as `work_queue_convert`. After the transcoding is complete, the script terminates the pool and waits five seconds to ensure that all the workers are terminated properly; otherwise, the workers will not finish cleaning themselves up before the next trial if the script does not wait.

4 Evaluation

This section analyzes the results of our benchmarks and evaluates the performances of our system. Table 1 shows averages of all the speedup results for every configuration we tested.

Based on our results, it is clear that the time needed to convert a given number of files is reduced as more WorkQueue workers are added unless the number of workers exceeds the number of files. Given a set of ten files, the speedup increases until the number of workers exceeds the number of files. At this point, times begin to be erratic, with most trials showing improvement but a small number of individual trials manifesting a very large increase in time needed, greatly reducing overall speedup. In general, increasing the number workers lead to speedups, albeit not linearly.

File Size	**# of Files**	**# of Workers**						
		1	2	4	8	16	24	30
15KB	10	1x	1.47x	1.56x	2.13x	1.85x	2.00x	**2.40x**
	100	1x	1.60x	2.80x	4.43x	5.96x	6.42x	**6.44x**
	1000	1x	1.65x	3.12x	5.02x	7.97x	9.27x	**9.31x**
1MB	10	1x	1.65x	2.40x	2.78x	3.05x	3.73x	**3.87x**
	100	1x	2.10x	3.87x	6.55x	**9.56x**	7.65x	8.27x
	1000	1x	2.17x	4.28x	7.75x	11.2x	10.5x	**12.12x**
10MB	10	1x	1.84x	2.46x	2.88x	**4.48x**	3.43x	3.27x
	100	1x	1.98x	3.90x	4.95x	**7.34x**	4.61x	4.76x
	1000	1x	1.74x	3.97x	5.63x	**6.26x**	4.75x	4.93x

Table 1: Benchmark Speedups Results.

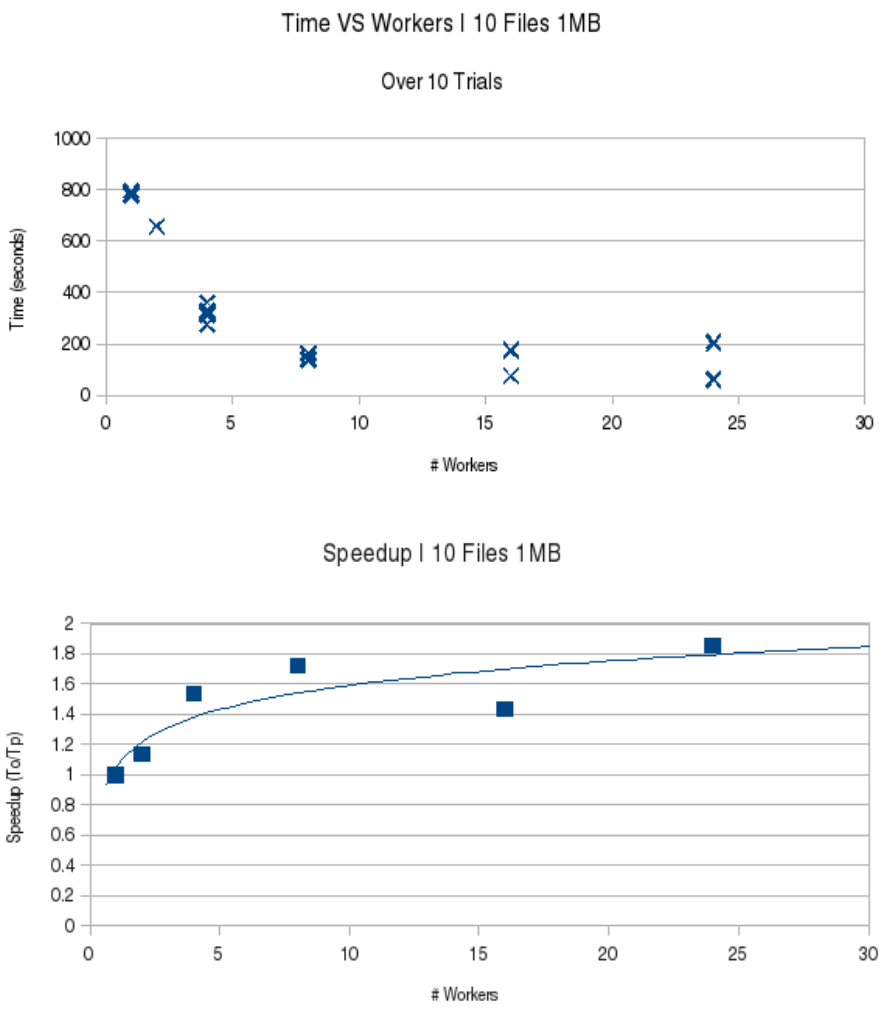

Figure 6: Execution Times and Performance Speedup for 10 1MB files.

Figure 6 shows execution times and performance speedup results for 10 1MB files. As the graphs show, when the number workers exceeds the number of files, the execution times stop decreasing in all trials. In a small number of trials, execution times are actually significantly increased. This leads to a great decrease in average performance speedup as more workers are added above the number of files. This erratic behaviour is mostly due to idiosyncrasies within the WorkQueue scheduler and the timing in which workers connect to the master.

The performance speedup graphs in Figure 7 demonstrate a closely logarithmic increase in speed as workers are added to the task of transcoding 100 and 1000 1MB files. Similarities between the two diagrams also demonstrate the consistency in behavior given large numbers of moderately sized files. The two figures show very similar behavior in relation to the best fit curve, with the 1000 files figure simply scaled up, demonstrating the continued performance enhancement of additional workers.

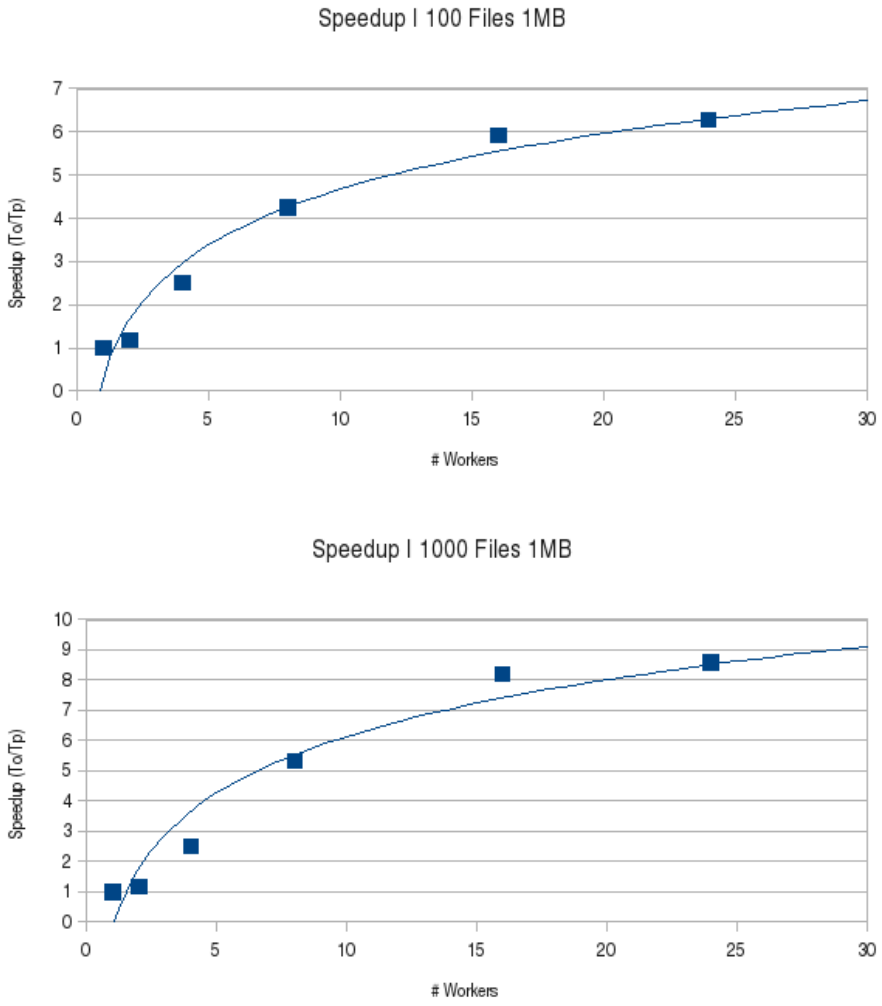

Figure 7: Performance Speedup for 100 and 1000 1MB files.

With sets of one hundred files, speedup will continue to improve as more workers are added. Figure 7 shows a contrary result where the speedup gained by adding a new worker is reduced as more workers are added. This is likely due to the time needed to transfer data across a network. As file sizes increase, the speedup gained by adding more workers decreases. If ten megabyte files are used, speedup will decrease after sixteen workers are added. This is likely caused by network transfer speeds being unable to cope with such large files. Moreover, because we utilize a master-worker paradigm and we transfer files from the master, it is likely that the master becomes a bottleneck when we benchmark larger number of files and larger sized files.

In summary, our benchmark results show that the image transcoding system we developed does indeed scale with the number of workers, albeit non-linearly. For medium numbers and sizes of files, we see a good amount of speedup as the number of workers increases.

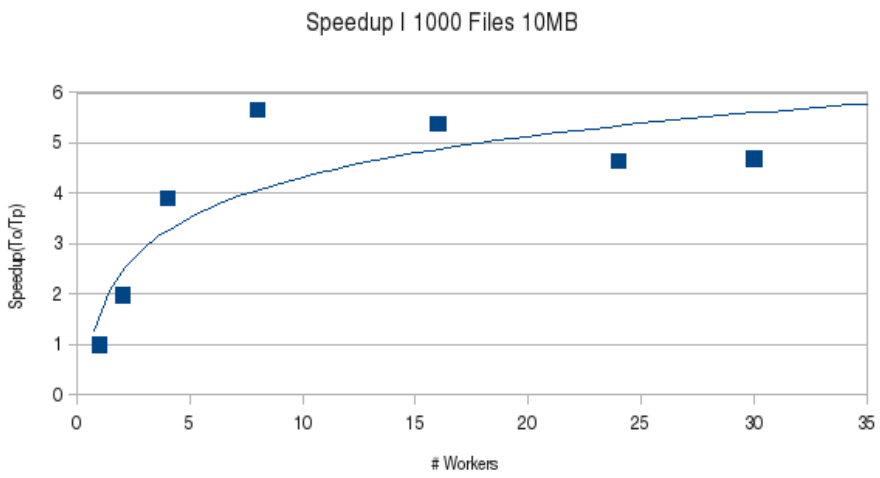

Figure 8: Performance Speedup for 1000 1MB files.

For small file sizes and small number of files we see only slight improvements, while for large large file sizes and large number of files we witness erratic behavior most likely due to network transfer bottlenecks.

5 Related Work

As noted in the introduction, our transcoding system design is heavily inspired by MapReduce [4] and Hadoop [5]. We decided against using Hadoop for the following reasons:

1. We did not need the full MapReduce programming model. Instead we simply needed a *Map* abstraction to form media transcoding workflows.

2. We did not have the ability to configure a full Hadoop cluster. Therefore, we utilized Python-WorkQueue [3] as a light-weight distributed computing framework that would work on our existing Condor [8] cluster.

3. We wanted the experience of building a distributed computing abstraction rather than utilizing a pre-existing solutions.

Architecturally, the master-worker model is not new [7], but it is effective and fits well with our pattern of computation. Similar systems have been built in the past and are in use today, but we wished to build a transcoding system ourselves and to analyze its performance.

6 Conclusion

Overall, our transcoding application was able to effectively reduce the amount of time required to transcode a large number of image files of varying sizes. Unfortunately, the amount of performance increase diminishes after a certain number of workers, meaning

that simply adding more machines to the application will not yield continued speedup. This demonstrates that while our system scales with the number of files and workers, there are limits to this scalability that must be considered in determining an optimal system configuration.

While the system was designed for image transcoding, it can also be modified for other applications as well. Because the system is designed to use a map to apply the same action to multiple files, it can be easily modified for alternative applications. The system could be modified to replace `convert` with a different program so that a different task could be applied to a set of files. Alternatively, the application could be modified to allow more flexible input of files, currently simply working on all valid files in a given directory. Our future goal is to generalized our image transcoding application to provided a general *Map* abstraction tool that can be utilized in many different contexts.

References

[1] ImageMagick. http://www.imagemagick.org/, 2013.

[2] H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[3] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain. Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications. In *Workshop on Python for High Performance and Scientific Computing at SC11*, 2011.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation*, 2004.

[5] Hadoop. http://hadoop.apache.org/, 2007.

[6] A. J. Hey, S. Tansley, and K. M. Tolle. *The fourth paradigm: data-intensive scientific discovery*. Microsoft Research Redmond, WA, 2009.

[7] J. Linderoth, S. Kulkarni, J.-P. Goux, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *IEEE High Performance Distributed Computing*, pages 43–50, Pittsburgh, Pennsylvania, August 2000.

[8] D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In F. Berman, A. Hey, and G. Fox, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.

[9] A. Thrasher, R. Carmichael, P. Bui, L. Yu, D. Thain, and S. Emrich. Taming complex bioinformatics workflows with Weaver, Makeflow, and Starch. In *Workflows in Support of Large-Scale Science (WORKS), 2010 5th Workshop on*, pages 1 –6, 2010.