zymake: a computational workflow system for machine learning and natural language processing

Eric Breck
Department of Computer Science
Cornell University
Ithaca, NY 14853
USA
ebreck@cs.cornell.edu

Abstract

Experiments in natural language processing and machine learning typically involve running a complicated network of programs to create, process, and evaluate data. Researchers often write one or more UNIX shell scripts to "glue" together these various pieces, but such scripts are suboptimal for several reasons. Without significant additional work, a script does not handle recovering from failures, it requires keeping track of complicated filenames, and it does not support running processes in parallel. In this paper, we present zymake as a solution to all these problems. zymake scripts look like shell scripts, but have semantics similar to makefiles. Using zymake improves repeatability and scalability of running experiments, and provides a clean, simple interface for assembling components. A zymake script also serves as documentation for the complete workflow. We present a zymake script for a published set of NLP experiments, and demonstrate that it is superior to alternative solutions, including shell scripts and makefiles, while being far simpler to use than scientific grid computing systems.

1 Introduction

Running experiments in natural language processing and machine learning typically involves a complicated network of programs. One program might extract data from a raw corpus, others might preprocess it with various linguistic tools, before finally the main program being tested is run. Further programs must evaluate the output, and produce graphs and tables for inclusion in papers and presentations. All of these steps can be run by hand, but a more typical approach is to automate them using tools such as UNIX shell scripts. We argue that any approach should satisfy a number of basic criteria.

Reproducibility At some future time, the original researcher or other researchers ought to be able to re-run the set of experiments and produce identical results[1]. Such reproducibility is a cornerstone of scientific research, and ought in principle to be easier in our discipline than in a field requiring physical measurements such as physics or chemistry.

Simplicity We want to create a system that we and other researchers will find easy to use. A system which requires significant overhead before any experiment can be run can limit a researcher's ability to quickly and easily try out new ideas.

A realistic life-cycle of experiments A typical experiment evolves in structure as it goes along - the researcher may choose partway through to add new datasets, new ranges of parameters, or new sets of models to test. Moreover, a computational experiment rarely works correctly the first time. Components break for various reasons, a tool may not perform as expected, and so forth. A usable tool must be simple to use in the face of such repeated re-execution.

Software engineering Whether writing shell scripts, makefiles, or Java, one is writing code, and software engineering concerns apply. One key principle is modularity, that different parts of

[1] User input presents difficulties which we will not discuss.

Software Engineering, Testing, and Quality Assurance for Natural Language Processing, pages 5–13,
Columbus, Ohio, USA, June 2008.

training regime	classes
two-way distinction	A vs B+O
two-way distinction	B vs A+O
three-way distinction	A vs B vs O
baseline comparison	A+B vs O

Table 1: Training regimes

a program should be cleanly separated. Another is generality, creating solutions that are re-usable in different specific cases. A usable tool must encourage good software engineering.

Inherent support for the combinatorial nature of our experiments Experiments in natural language processing and machine learning typically compare different datasets, different models, different feature sets, different training regimes, and train and test on a number of cross-validation folds. This produces a very large number of files which any system must handle in a clean way.

In this paper, we present `zymake`[2], and argue that is superior to several alternatives for the task of automating the steps in running an experiment in natural language processing or machine learning.

2 A Typical NLP Experiment

As a running example, we present the following set of experiments (abstracted from (Breck et al., 2007)). The task is one of entity identification - we have a large dataset in which two different types of opinion entities are tagged, type A, and type B. We will use a sequence-based learning algorithm to model the entities, but we want to investigate the relationship between the two types. In particular, will it be preferable to learn a single model which predicts both entity type A and entity type B, or two separate models, one predicting A, and one predicting B. The former case makes a three-way distinction between entities of type A, of type B, and of type O, all other words. The latter two models make a distinction between type A and both other types or between type B and both other types. Furthermore, prior work to which we wish to compare does not distinguish at all between type A and type B, so we also need a model which just predicts entities to be of either type A or B, versus the background O. These four training regimes are summarized in Table 1.

Given one of these training regimes, the model is trained and tested using 10-fold cross-validation, and the result is evaluated using precision and recall. The evaluation is conducted separately for class A, for class B, and for predicting the union of both classes.

2.1 Approach 1: A UNIX Shell Script

Many researchers use UNIX shell scripts to co-ordinate experiments[3]. Figure 1 presents a potential shell script for the experiments discussed in Section 2. Shell scripting is familiar and widely used for co-ordinating the execution of programs. However, there are three difficulties with this approach - it is difficult to partially re-run, the specification of the filenames is error-prone, and the script is badly modularized.

Re-running the experiment The largest difficulty with this script is how it handles errors - namely, it does not. If some early processes succeed, but later ones fail, the researcher can only re-run the entire script, wasting the time spent on the previous run. There are two common solutions to this problem. The simplest is to comment out the parts of the script which have succeeded, and re-run the script. This is highly brittle and error-prone. More reliable but much more complicated is to write a wrapper around each command which checks whether the outputs from the command already exist before running it. Neither of these is desirable. It is also worth noting that this problem can arise not just through error, but when an input file changes, an experiment is extended with further processing, additional graphs are added, further statistics are calculated, or if another model is added to the comparison.

[2]Any name consisting of a single letter followed by `make` already refers to an existing software project. `zymake` is the first pronouncable name consisting of a two letter prefix to `make`, starting from the end of the alphabet. I pronounce "zy-" as in "zydeco."

[3]Some researchers use more general programming languages, such as Perl, Python, or Java to co-ordinate their experiments. While such languages may make some aspects of co-ordination easier – for example, such languages would not have to call out to an external program to produce a range of integers as does the script in Figure 1 – the arguments that follow apply equally to these other approaches.

```
for fold in `seq 0 9`; do
  extract-test-data $fold raw-data $fold.test
  for class in A B A+B; do
    extract-2way-training $fold raw-data $class > $fold.$class.train
    train $fold.$class.train > $fold.$class.model
    predict $fold.$class.model $fold.test > $fold.$class.out
    prep-eval-2way $fold.$class.out > $fold.eval-in
    eval $class $fold.$class.eval-in > $fold.$class.eval
  done
  extract-3way-training $fold raw-data > $fold.3way.train
  train $fold.3way.train > $fold.3way.model
  predict $fold.3way.model $fold.test > $fold.3way.out
  for class in A B A+B; do
    prep-eval-3way $class $fold.3way.out > $fold.3way.$class.eval-in
    eval $class $fold.3way.$class.eval-in > $fold.3way.$class.eval
  done
done
```

Figure 1: A shell script

Problematic filenames In this example, a filename is a concatenation of several variable names - e.g. `$(fold).$(class).train`. This is also error-prone - the writer of the script has to keep track, for each filename, of which attributes need to be specified for a given file, and the order in which they must be specified. Either of these can change as an experiment's design evolves, and subtle design changes can require changes throughout the script of the references to many filenames.

Bad modularization In this example, the `eval` program is called twice, even though the input and output files in each case are of the same format. The problem is that the filenames are such that the line in the script which calls `eval` needs to be include information about precisely which files (in one case `$fold.3way.$class`, and in the other `$fold.$class`) are being evaluated. This is irrelevant – a more modular specification for the `eval` program would simply say that it operates on a `.eval-in` file and produces an `.eval` file. We will see ways below of achieving exactly this.[4]

[4]One way of achieving this modularization with shell scripts could involve defining functions. While this could be effective, this greatly increases the complexity of the scripts.

```
%.model: %.train
  train $< > $@

%.out: %.model %.test
  predict $^ > $@
```

Figure 2: A partial makefile

2.2 Approach 2: A `makefile`

One solution to the problems detailed above is to use a makefile instead of a shell script. The `make` program (Feldman, 1979) bills itself as a "utility to maintain groups of programs"[5], but from our perspective, `make` is a declarative language for specifying dependencies. This seems to be exactly what we want, and indeed it does solve some of the problems detailed above. `make` has several new problems, though, which result in its being not an ideal solution to our problem.

Figure 2 presents a portion of a makefile for this task. For this part, the makefile ideally matches what we want. It will pick up where it left off, avoiding the re-running problem above. The question of filenames is sidestepped, as we only need to deal with the extensions here. And each command is neatly

[5]GNU `make` manpage.

partitioned into its own section, which specifies its dependencies, the files created by each command, and the shell command to run to create them. However, there are three serious problems with this approach.

Files are represented by strings The first problem can be seen by trying to write a similar line for the `eval` command. It would look something like this:

```
%.eval: %.eval-in
  eval get-class $^ > $@
```

However, it is hard to write the code represented here as `get-class`. This code needs to examine the filename string of `$^` or `$@`, and extract the class from that. This is certainly possible using standard UNIX shell tools or make extensions, but it is ugly, and has to be written once for every time such a field needs to be accessed. For example, one way of writing `get-class` using GNU make extensions would be:

```
GETCLASS = $(filter A B A+B,\
$(subst ., ,$(1)))

%.eval: %.eval-in
  eval $(call GETCLASS,$@) $^ > $@
```

The basic problem here is that to `make`, a file is represented by a string, its filename. For machine learning and natural language processing experiments, it is much more natural to represent a file as a set of key-value pairs. For example, the file `0.B.model` might be represented as { `fold = 0, class = B, filetype = model` } .

Combinatorial dependencies The second problem with `make` is that it is very difficult to specify combinatorial dependencies. If one continued to write the makefile above, one would eventually need to write a final `all` target to specify all the files which would need to be built. There are 60 such files: one for each fold of the following set

```
$fold.3way.A.eval
$fold.3way.B.eval
$fold.3way.A+B.eval
$fold.A.eval
$fold.B.eval
$fold.A+B.eval
```

```
%.taggerA.pos: %.txt
  tagger_A $^ > $@

%.taggerB.pos: %.txt
  tagger_B $^ > $@

%.taggerC.pos: %.txt
  tagger_C $^ > $@

%.chunkerA.chk: %.pos
  chunker_A $^ > $@

%.chunkerB.chk: %.pos
  chunker_B $^ > $@

%.chunkerC.chk: %.pos
  chunker_C $^ > $@

%.parserA.prs: %.chk
  parser_A $^ > $@

%.parserB.prs: %.chk
  parser_B $^ > $@

%.parserC.prs: %.chk
  parser_C $^ > $@
```

Figure 3: A non-functional makefile for testing three independent decisions

There is no easy way in `make` of listing these 60 files in a natural manner. One can escape to a shell script, or use GNU `make`'s `foreach` function, but both ways are messy.

Non-representable dependency structures The final problem with `make` also relates to dependencies. It is more subtle, but it turns out that there are some sorts of dependency structures which cannot be represented in `make`. Suppose I want to compare the effect of using one of three parsers, one of three part-of-speech-taggers and one of three chunkers for a summarization experiment. This involves three separate three-way distinctions in the makefile, where for each, there are three different commands that might be run. A non-working example is in Fig-

ure 3. The problem is that `make` pattern rules (rules using the `%` character) can only match the suffix or prefix of a filename[6]. This makefile does not work because it requires the parser, chunker, and tagger to all be the last part of the filename before the type suffix.

2.3 Approach 3: `zymake`

`zymake` is designed to address the problems outlined above. The key principles of its design are as follows:

- Like `make`, `zymake`files can be re-run multiple times, each time picking up where the last left off.
- Files are specified by key-value sets, not by strings
- `zymake` includes a straightforward way of handling combinatorial sets of files.
- `zymake` syntax is minimally different from shell syntax.

Figure 4 presents a zymakefile which runs the running example experiment. Rather than explaining the entire file at once, we will present a series of increasingly complex parts of it.

Figure 5 presents the simplest possible zymakefile, consisting of one *rule*, which describes how to create a `$().test` file, and one *goal*, which lists what files should be created by this file. A rule is simply a shell command[7], with some number of *interpolations*[8]. An *interpolation* is anything between the characters `$(` and the matching `)`. This is the only form of interpolation done by `zymake`, so as to minimally conflict with other interpolations done by the shell, scripting languages such as Perl, etc.

[6]Thus, if we were only comparing two sets of items – e.g. parsers and taggers but not chunkers – we could write this set of dependencies by using a prefix to distinguish one set and a suffix to distinguish the other. This is hardly pretty, though, and does not extend to more than two sets.

[7]Users who are familiar with UNIX shells will find it useful to be able to use input/output redirection and pipelines in zymakefiles. Knowledge of advanced shell programming is not necessary to use `zymake`, however.

[8]This term is used in Perl; it is sometimes referred to in other languages as "substitution" or "expansion."

```
extract-test-data $(fold) raw-data
 $(>).test

extract-2way-training $(fold) raw-data
 $(class) > $(train="2way").train

extract-3way-training $(fold) raw-data
  > $(train="3way").train

train $().train > $().model

predict $().model $().test > $().out

prep-eval-3way $(class) $().out >
  $(train="3way").eval-in

prep-eval-2way $().out >
  $(train="2way").eval-in

eval $(class) $().eval-in > $().eval

classes = A B A+B
ways = 2way 3way

: $(fold = *(range 0 9)
    class = *classes
    train = *ways).eval
```

Figure 4: An example zymakefile. The exact commands run by this makefile are presented in Appendix A.

```
extract-test-data raw-data $(>).test

: $().test
```

Figure 5: Simple zymakefile #1

```
extract-test-data $(fold) raw-data
 $(>).test

: $(fold=0).test $(fold=1).test
```

Figure 6: Simple zymakefile #2

```
extract-test-data $(fold) raw-data
$(>).test

folds = 0 1

: $(fold=*folds).test
```

Figure 7: Simple zymakefile #3

The two interpolations in this example are file interpolations, which are replaced by `zymake` with a generated filename. Files in `zymake` are identified not by a filename string but by a set of key-value pairs, along with a suffix. In this case, the two interpolations have no key-value pairs, and so are only represented by a suffix. Finally, there are two kinds of file interpolations - *inputs*, which are files that are required to exist before a command can be run, and *outputs*, which are files created by a command[9]. In this case, the interpolation `$(>).test` is marked as an output by the > character[10], while `$().test` is an input, since it is unmarked.

The goal of this program is to create a file matching the interpolation `$().test`. The single rule does create a file matching that interpolation, and so this program will result in the execution of the following single command:

```
extract-test-data raw-data .test
```

Figure 6 presents a slightly more complex zymakefile. In this case, there are two goals - to create a `.test` file with the key `fold` having the value `0`, and another `.test` file with `fold` equal to `1`. We also see that the rule has become slightly more complex – there is now another interpolation. This, however, is not a file interpolation, but a variable interpolation. `$(fold)` will be replaced by the value of `fold`.

[9] Unlike `make`, `zymake` requires that each command explicitly mention an interpolation corresponding to each input or output file. This restriction is caused by the merging of the command part of the rule with the dependency part of the rule, which are separate in `make`. We felt that this reduced redundancy and clutter in the zymakefiles, but this may occasionally require writing a wrapper around a program which does not behave in this manner.

[10] `zymake` will also infer that any file interpolation following the > character, representing standard output redirection in the shell, is an output

Executing this zymakefile results in the execution of two commands:

```
extract-test-data 0 raw-data 0.test
extract-test-data 1 raw-data 1.test
```

Note that the output files are now not just `.test` but include the fold number in their name. This is because `zymake` infers that the fold key, mentioned in the extract rule, is needed to distinguish the two test files. In general the user should specify as few keys as possible for each file interpolation, and allow `zymake` to infer the exact set of keys necessary to distinguish each file from the rest[11].

Figure 7 presents a small refinement to the zymakefile in Figure 6. The commands that will be run are the same, but instead of separately listing the two test files to be created, we create a variable `folds` which is a list of all the folds we want, and use a *splat* to create multiple goals. A splat is indicated by the asterisk character, and creates one copy of the file interpolation for each value in the variable's list.

Figure 4 is now a straightforward extension of the example we have seen so far. It uses a few more features of `zymake` that we will not discuss, such as string-valued keys, and the `range` function, but further documentation is available on the `zymake` website. `zymake` wants to create the goals at the end, so it examines all the rules and constructs a *directed acyclic graph*, or *DAG*, representing the dependencies among the files. It then executes the commands in some order based on this DAG – see Section 3 for discussion of execution order.

2.4 Benefits of `zymake`

`zymake` satisfies the criteria set out above, and handles the problems discussed with other systems.

- *Reproducibility*. By providing a single file which can be re-executed many times, `zymake` encourages a development style that encodes all information about a workflow in a single file. This also serves as documentation of the complete workflow.

[11] Each file will be distinguished by all and only the keys needed for the execution of the command that created it, and the commands that created its inputs. A unique, global ordering of keys is used along with a unique, global mapping of filename components to key, value pairs so that the generated filename for each file uniquely maps to the appropriate set of key, value pairs.

- *Simplicity*. zymake only requires writing a set of shell commands, annotated with interpolations. This allows researchers to quickly and easily construct new and more complex experiments, or to modify existing ones.
- *Experimental life-cycle*. zymake can re-execute the same file many times when components fail, inputs change, or the workflow is extended.
- *Software engineering*. Each command in a zymakefile only needs to describe the inputs and outputs relevant for that command, making the separate parts of the file quite modular.
- *Combinatorial experiments*. zymake includes a built-in method for specifying that a particular variable needs to range over several possibilities, such as a set of models, parameter values, or datasets.

2.5 Using zymake

Beginning to use zymake is as simple as downloading a single binary from the website[12]. Just as with a shell script or makefile, the user then writes a single textual zymakefile, and passes it to zymake for execution. Typical usage of zymake will be in an edit-run development cycle.

3 Parallel Execution

For execution of very large experiments, efficient use of parallelism is necessary. zymake offers a natural way of executing the experiment in a maximally parallel manner. The default serial execution does a topological sort of the DAG, and executes the components in that order. To execute in parallel, zymake steps through the DAG starting at the roots, starting any command which does not depend on a command which has not yet executed.

To make this practical, of course, remote execution must be combined with parallel execution. The current implementation provides a simple means of executing a remote job using ssh, combined with a simple /proc-based measure of remote cpu utilization to find the least-used remote cpu from a provided set. We are currently looking at extending zymake to interface it with the Condor system (Litzkow et al., 1988). Condor's DAGMan is designed to execute a DAG in parallel on a set of remote machines, so it should naturally fit with zymake. Interfaces to other cluster software are possible as well. Another important extension will be to allow the system to throttle the number of concurrent jobs produced and/or collect smaller jobs together, to better match the available computational resources.

[12]Binaries for Linux, Mac OS X, and Windows, as well as full source code, are available at http://www.cs.cornell.edu/~ebreck/zymake/.

4 Other approaches

Deelman et al. (2004) and Gil et al. (2007) describe the Pegasus and Wings systems, which together have a quite similar goal to zymake. This system is designed to manage large scientific workflows, with both data and computation distributed across many machines. A user describes their available data and resources in a semantic language, along with an abstract specification of a workflow, which Wings then renders into a complete workflow DAG. This is passed to Pegasus, which instantiates the DAG with instances of the described resources and passes it to Condor for actual execution. The system has been used for large-scale scientific experiments, such as earthquake simulation. However, we believe that the added complexity of the input that a user has to provide over zymake's simple shell-like syntax will mean a typical machine learning or natural language processing researcher will find zymake easier to use.

The GATE and UIMA architectures focus specifically on the management of components for language processing (Cunningham et al., 2002; Ferrucci and Lally, 2004). While zymake knows nothing about the structure of the files it manages, these systems provide a common format for textual annotations which all components must use. GATE provides a graphical user interface for running components and for viewing and producing annotations. UIMA provides a framework not just for running experiments but for data analysis and application deployment. Compared to writing a zymake script, however, the requirements for using these systems to manage an experiment are greater. In addition, both these architectures most naturally support com-

ponents written in Java (and in the case of UIMA, C++). zymake is agnostic as to the source language of each component, making it easier to include programs written by third parties or by researchers who prefer different languages.

make, despite dating from 1979, has proved its usefulness over time, and is still widely used. Many other systems have been developed to replace it, including ant[13], SCons[14], maven[15], and others. However, so far as we are aware, none of these systems solves the problems we have described with make. As with make and shell scripts, running experiments is certainly possible using these other tools, but we believe they are far more complex and cumbersome than zymake.

5 Future Extensions

There are a number of extensions to zymake which could make it even more useful. One is to allow the dependency DAG to vary during the running of the experiment. At the moment, zymake requires that the entire DAG be known before any processes can run. As an example of when this is less than ideal, consider early-stopping an artificial neural network. One way of doing this is train the network to full convergence, and output predictions from the intermediate networks at some fixed interval of epochs. We would like then to evaluate all these predictions on held-out data (running one process for each of them) and then to choose the point at which this score is maximized (running one process for the whole set). Since the number of iterations to convergence is not known ahead of time, at the moment we cannot support this structure in zymake. We plan, however, to allow the structure of the DAG to vary at run-time, allowing such experiments.

We are also interested in other extensions, including an optional textual or graphical progress bar, providing a way for the user to have more control over the string filename produced from a key-value set[16], and keeping track of previous versions of created files, to provide a sort of version control of the output files.

[13] http://ant.apache.org/.

[14] http://www.scons.org/.

[15] http://maven.apache.org/.

[16] This will better allow zymake to interact with other workflows.

6 Conclusion

Most experiments in machine learning and natural language processing involve running a complex, interdependent set of processes. We have argued that there are serious difficulties with common approaches to automating these experiments. In their place, we offer zymake, a new scripting language with shell-like syntax but make-like semantics. We hope our community will find it as useful as we have.

Acknowledgements

We thank Yejin Choi, Alex Niculescu-Mizil, David Pierce, the Cornell machine learning discussion group, and the anonymous reviewers for helpful comments on earlier drafts of this paper.

A Output of Figure 4

We present here the commands run by zymake when presented with the file in Figure 4. We present only the commands run for fold 0, not for all 10 folds. Also, in actual execution zymake adds a prefix to each filename based on the name of the zymakefile, so as to separate different experiments. Finally, note that this is only one possible order that the commands could be run in.

```
extract-2way-training 0 raw-data A > A.0.2way.train
train A.0.2way.train > A.0.2way.model
extract-2way-training 0 raw-data B > B.0.2way.train
train B.0.2way.train > B.0.2way.model
extract-2way-training 0 raw-data A+B > AB.0.2way.train
train AB.0.2way.train > AB.0.2way.model
extract-3way-training 0 raw-data > 0.3way.train
train 0.3way.train > 0.3way.model
extract-test-data 0 raw-data 0.test
predict A.0.2way.model 0.test > A.0.2way.out
prep-eval-2way A.0.2way.out > A.0.2way.eval-in
eval A A.0.2way.eval-in > A.0.2way.eval
predict B.0.2way.model 0.test > B.0.2way.out
prep-eval-2way B.0.2way.out > B.0.2way.eval-in
eval B B.0.2way.eval-in > B.0.2way.eval
predict AB.0.2way.model 0.test > AB.0.2way.out
prep-eval-2way AB.0.2way.out > AB.0.2way.eval-in
eval A+B AB.0.2way.eval-in > AB.0.2way.eval
predict 0.3way.model 0.test > 0.3way.out
prep-eval-3way A 0.3way.out > A.0.3way.eval-in
eval A A.0.3way.eval-in > A.0.3way.eval
prep-eval-3way B 0.3way.out > B.0.3way.eval-in
eval B B.0.3way.eval-in > B.0.3way.eval
prep-eval-3way A+B 0.3way.out > AB.0.3way.eval-in
eval A+B AB.0.3way.eval-in > AB.0.3way.eval
```

References

Eric Breck, Yejin Choi, and Claire Cardie. 2007. Identifying expressions of opinion in context. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-2007)*, Hyderabad, India, January.

Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. 2002. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL '02)*, Philadelphia, July.

Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. 2004. Pegasus : Mapping scientific workflows onto the grid. In *Across Grids Conference*, Nicosia, Cyprus.

Stuart I. Feldman. 1979. Make-a program for maintaining computer programs. *Software - Practice and Experience*, 9(4):255–65.

David Ferrucci and Adam Lally. 2004. UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Nat. Lang. Eng.*, 10(3-4):327–348.

Yolanda Gil, Varun Ratnakar, Ewa Deelman, Gaurang Mehta, and Jihie Kim. 2007. 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)*, Vancouver, British Columbia, Canada, July.

Michael Litzkow, Miron Livny, and Matthew Mutka. 1988. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June.