GOLEMIZE.PY USAGE MANUAL

Adam B. Norberg

August 12, 2011

Contents

CHAPTER 1

INTRODUCTION

The Python programming language has many merits, but support for cross-machine parallel processing is not one of them. A process-centric job distribution engine, Golem, has been written by Ryan Bressler in part to allow Python scripts to be conveniently launched on every machine in a cluster, but it provides no help in dividing the input into separate tasks or in aggregating the results, other than collating individual lines of standard output in random order.

`golemize.py` is a module intended to provide these features to Python scripts that would like to use the Golem framework to perform parallel computation. It allows a sequence of inputs and a function to perform computation to be distributed across a computational cluster, with different inputs sent to different machines to run in parallel. It is saving the input and output data in the `pickle` format to allow it to be loaded on the worker nodes or sent from the worker nodes to the client, but hides most of the complexity of doing so from the user of the module.

Chapter 2

API

`golemize.py` contains its main “Golemizer” class, two exception classes, and assorted functions that are not inside one of its classes. They are used to hold and represent settings for connecting to a Golem server and to perform the actual distributed computation.

2.1 class Golemizer

This is the main class that connects to a remote Golem server, distributes the input into tasks, waits for the job to complete, and sets up a Generator that aggregates individual results.

2.1.1 Initializer

The initializer constructs a ready-to-use Golemizer object from server connection and file storage information. The initializer performs little validation, so inappropriate parameters here are more likely to show up as bad behavior later on rather than errors immediately.

Parameters

serverURL A string representing a URL that can reach the Golem master server.

serverPass The password for job-write access to the server.

golemOutputPath The *local file* path representing the root output directory for all Golem nodes. This is the directory that itself contains a bunch of directories named "golem_01", "golem_02",... "golem_*nn*". `golemize.py` is aware of this structure and will stop working if Golem changes it.

golemIdSeq A sequence of `int` representing the set of Golem machines that exist. This will presumably be the output of `range(1, N+1)`, where N is defined to be the number of worker nodes in the cluster. A previous version of the design specified low and high machine ranges, but since `range` already expresses that gracefully and would be used anyway, there seems to be no reason to be so specific.

pickleScratch A string representing the local file path, *which must be visible to and the same for all the worker nodes in the Golem cluster*, to which `golemize.py` should write the (large number of) intermediate files it produces- the files loaded by the nodes when they execute the computation function.

thisLibraryPath A string representing the full path to where `golemize.py` is installed on the worker nodes.

pyPath A string representing the path to the Python binary on the worker nodes. Defaults to `/hpc/bin/python`, which is correct for the `glados` cluster set up at the Institute for Systems Biology.

pickleOut An *optional* string representing the directory where worker nodes should put the files holding the results of the computations. If `None`, the Golem working directory is used, which is almost surely correct unless you would explicitly like the nodes to write to a different result share (generally because the pickled results are intended to be kept for future reference).

taskSize An `int` representing the number of inputs to assign per Golem task. Defaults to 10. Most likely to be changed after initialization, since a Golemizer is most likely to be constructed straight off a config file, but this is likely to need to change for individual programs using Golemizer.

2.1.2 Fields

Any of the configuration settings of a Golemizer can be changed after construction. Use caution; Golemizer performs little validation of its own state.

masterPath The canonical path to the Golem master server being connected to. This is not the same as the serverURL parameter; it was run through `golem.canonizeMaster` first.

serverPass The password to connect to the server. serverPass parameter to the initializer.

golemOutPath Root path to Golem working directories. golemOutputPath parameter to the initializer.

golemIds List of strings containing the ID numbers of Golem workers in use. This includes the leading 0s on single-digit IDs.

pickleInputShare Share to put pickled input data on. pickleScratch parameter to the initializer.

pyPath Path to the workers' Python interpreters. pyPath parameter to the initializer.

thisLibraryPath Where the workers should find `golemize.py`. thisLibraryPath parameter to the initializer.

jobOutputPath Where the workers should, from their perspective, write output. `None` or an empty string in the jobOutputPath is translated to the current directory before here, so this field shouldn't be None.

taskSize Number of lines per task. Inititalized via taskSize. Most likely to be rewritten during the course of normal use.

2.1.3 setTaskSize

A convenience method to set the taskSize field. It exists mostly as a reminder that the taskSize field exists and is expected to be changed as-needed, rather than left as configured.

Input

`value`–an integer with the number of inputs that should be bundled into a task.

Output

`None`.

Error handling

None. This function cannot throw an exception. If Input is not an integer, this will cause a call to `goDoIt` to crash; it is not detected sooner.

2.1.4 _spill

Internal function (which doesn't need to be an instance function, actually). Writes `nextList` to a new file named `pickleCount`.pkl in the fast binary format, then closes the file.

Input

nextList Object to serialize. As used, this is always a List.

pickleCount ID number (or other key, actually) of pickle to generate and save with a .pkl extension in the current working directory.

Output

`None`. Produces a file on disk in the current working directory.

Error handling

Does not handle any of the many errors that `cPickle.dump` can produce. Notably, if the list contains data that cannot be pickled, this will give a related crash.

2.1.5 goDoIt

The primary purpose of a Golemizer. Takes a provided calculation function and its inputs, distributes those inputs into temporary files, and runs the provided calculation on the worker nodes using the provided inputs. Finally, it collects the results and returns a generator that iterates over them one at a time, so it does not need to be possible to fit all the results in main memory at once.

Input

`goDoIt` has three required parameters and three optional parameters. These represent the two sources of data, the function itself, and options about how goDoIt should distribute the target code to the worker machines.

inputSeq Any sequence object that yields a inputs to be provided to the computation function. One input will be used per call to the function. `taskSize` inputs will be bundled per Golem task. This does not need to be a list; any object that can be drawn from in a for loop will suffice. The combination of all the data it generates does not need to fit in memory, but `taskSize` consecutive elements do.

commonData In addition to the per-task input, this is one input object that will be provided to every invocation of the calculation function. Use this for data needed for calculation that doesn't change per input; putting it here, instead of copying it as part of every element of `inputSeq`, saves (a potentially large amount of) disk space and serialization time. `None` is a legal value, if you don't need this feature.

targetFunction The function to perform calculations with. It must take exactly two parameters; its result will be saved and later returned as part of the sequence of results yielded by the output of `goDoIt`. The first parameter will be drawn from `inputSeq`; the second parameter will be the `commonData`. Remember when crafting your function that the input will be run in no guaranteed order, distributed arbitrarily to different machines, without the greater context of your script, so your function cannot rely on data that is not in either of its two inputs.

binplace *Defaults to True.* Whether or not `golemize.py` should copy your script into the same share it's placing the input files in and run it from there. If this evaluates as False, your script *must* be placed such that all worker machines can reach it. However, if this is true, your script must be contained in a single file. This is described in more detail in the "Discussion of Usage" chapter, under "The 'One File or Don't Binplace' rule".

alternateSource *Defaults to None.* If defined, this is where `golemize.py` should look for your script; otherwise, it will seek out whereever you launched it from. If you have a copy of the script already on the common path of all the worker nodes, but that's not where you launched this from, use both that and `binplace`, which will also be obeyed and run the script out of the location defined here. If `binplace` is true, the script defined here will be copied.

recursive *Defaults to False.* Should always remain False. At the time I wrote it, I thought there would be a case where you would want to shut off the safety check to make sure that you aren't about to launch a new set of Golem tasks from within a Golem task. Shutting this check off is supported in case you want to use Golem to break the problem down further in your solution algorithm. This turns out not to work, since Goelm will wait forever on your script to complete before it can free the worker node and assign the work you just scheduled upon it, which will never happen until that work is scheduled. If Golem somehow determines that a running program is idle and the worker can get additional work until the program wakes up, then this flag will become useful. Currently, don't touch it, no situation where it is on will result in a working program.

Output

A generator that produces the results of your calculation function, in the exact same order your input sequence ran in. It is not a list, so it cannot be directly indexed; if you need that, unpack it into a list. Results are not loaded any sooner than they need to be, and are not cached after they have been yielded by the generator, so result data sets that don't fit into RAM can still be processed.

If your calculation function throws an exception, then the generator will throw that exception during iteration at the point corresponding to the input that caused that exception. This will, unfortunately, stop the generator, so your calculation function should only throw exceptions if it encounters a real error that would make you want to stop processing the data set entirely.

Error handling

Any errors in your calculation function will be reraised by the Generator when it reaches the point corresponding to the input that caused the error.

If your input data cannot be pickled, `cPickle` will throw one of its errors, which goDoIt will not catch.

If you are attempting to call goDoIt from a task on a worker node generated by `golemize.py`, an `InfiniteRecursionError` will be thrown. If this is deliberate, turn on the `recursive` flag, but at the moment, this doesn't actually work, so please design your program to do something else instead.

If the job completes, but goDoIt cannot find all the results, an `ExecutionFailure` is thrown. Unfortunately, goDoIt can't tell what happened; check the standard error logs from the job.

2.2 Factory methods

These functions construct a `Golemizer`, designed to enable the use of configuration files without manually unpacking them into the initializer. These are convenience methods that do that unpacking based on a schema of name-value pairs, which are described in more detail in the chapter on Configuration.

2.2.1 dictToGolemizer

Constructs a Golemizer from a `dict` that fits the schema described in the Configuration chapter.

Input

`config`–a `dict` containing the key-value pairs specified in the Configuration chapter.

Output

A Golemizer described by that configuration data.

Error handling

If the input `dict` lacks a required field, a KeyError will be raised. Any exceptions in the Golemizer initializer will go uncaught.

2.2.2 jsonToGolemizer

Constructs a Golemizer from a file containing JSON data that, when interpreted as a `dict` using strings as keys, fits the schema described in the Configuration chapter.

Input

`jsonfile`–an open file handle to a file containing a JSON object representing the key-value pairs specified in the Configuration chapter.

Output

A Golemizer described by that configuration data.

Error handling

If the input file does not contain exactly one properly-formatted JSON object, the JSON parser will raise a relevant exception, which will go uncaught. If the input JSON object lacks a required field, a KeyError will be raised. Any exceptions in the Golemizer initializer will go uncaught.

2.3 Exception classes

These are simple, standard Exception classes used in case something goes wrong in `goDoIt`.

2.3.1 ExecutionFailure

Thrown if a task is detected as having failed to produce output.

2.3.2 InfiniteRecursionError

Thrown if goDoIt is detected as being called from within a process that is already a Golem task.

2.4 Functions intended for internal use

The following functions are not intended to be used except as utility functions for `goDoIt`. They are documented sparsely and may change without notice.

2.4.1 unpickleSequence

This is the generator that loads a list of result files, unpickles each of them in turn, and iterates over the results contained within.

2.4.2 jumpToTask

This loads a script file and executes a function within it after loading the inputs to that function. This is the function that actually executes on the worker nodes.

CHAPTER 3

DISCUSSON OF USAGE

This section contains important notes on how to use `golemize.py` with regard to its requirements beyond simple function prototypes.

3.1 Making your calculation function fit the prototype

The computation function provided to `Golemizer.goDoIt` must take exactly two parameters- one that's taken off the input list, with the other common to all invocations of the function. In many cases, your actual function will need input more complex than that, generally to hold more values.

Fortunately, Python is glad to help you here. Pack all the inputs to an invocation of the calculation function into a tuple, and unpack it at the beginning of the function. If you have prewritten code you'd like to not mutilate, create a wrapper function that does only this.

Similarly, if you have more than one data object common to all invocations of your function, pack it into a tuple or dict.

`goDoIt` does support functions that provide multiple outputs. A function that pretends to produce multiple outputs is actually producing a single tuple containing all of its results, and that tuple is itself added to the list of results. Multiple assignment is simply automatic unpacking of a tuple, and it works just as well out of a sequence of stored tuples as it does out of a function.

3.2 The “One File or Don’t Binplace” rule

`golemize.py` assumes you have better things to do than copy files around just because you’ve written a script you’d like to distribute. This is why it will move a file–“binplace”, in Microsoft parlance, which has forever corrupted my vocabulary–into the input dump directory for you: you might have your script file stored somewhere not visible to the Golem workers, and it’s glad to put it somewhere they can pick it up.

However, `golemize.py` only knows about one file: the one that contains your calculation function. It doesn’t know anything about any other files your script may rely on. If your function imports another script that you wrote, `golemize.py` knows absolutely nothing about it and will fail to carry it along. Importing your script will therefore fail, which you will discover as soon as you try to start iterating over results.

If your function calls into another script you wrote, that’s fine, but you’ll have to put them on a path visible to the Golem workers yourself, and tell `golemize` not to touch it. You can either have automatic binplacement or a multiple-file script, but not both. Set the `binplace` parameter in `goDoIt` to `False` to prevent an (incorrect) file copy in this case.

3.3 Always check __name__

In Python, there is a built-in variable called “`__name__`” that stores one of two things:

- The name of the Python file from which the code module being executed was imported
- The string “`__main__`”

The first of those is used if the module was imported and is being used as a library, while “`__main__`” is used if the module was directly executed from the command line. This is designed for you to have a way to check if your script is the main entry point for a program, because if it’s only being imported, it should not run the vast majority of its code.

`golemize.py` imports your script as a module so it can access your calculation function. If your program is not checking __main__, then it will try to run from the beginning with a very strange-looking command line, and will probably crash in the middle of import. Potentially worse is *not* crashing in the middle of import, which means your program has run against nonsensical input and is now presumably trying to split this work into Golem tasks and run it! `goDoIt` has a check for this to prevent an infinite storm of insensible jobs from flooding the cluster, but that still means your program doesn't work.

Always check that $\_\_name\_\_ == “\_\_main\_\_''$ before the main logic of your program, or it will attempt to run spuriously during import.

3.4 Unified paths

As currently implemented, `golemize.py` assumes that file paths are the same between where the script that launches it is running from and what the golem nodes will see. Here at ISB, as implemented, this means that everything you need pretty much has to be on /titan or /proj and you can only launch your script from a Linux server.

Sorry about that. If this turns out to be a big problem, I'll write significantly more elaborate file resolution logic involving the use of ssh into the code. That represents a surprisingly large volume of effort, so it didn't seem like a priority for this first version.

CHAPTER 4

CONFIGURATION

`golemize.py` needs to be aware of all the settings required to connect to a Golem server, including where it should put files for Golem workers. Settings such as this are expected to be loaded out of a JSON-formatted file. (The convenience function `jsonToGolemizer` has been provided to load such a settings file for you.)

4.1 JSON configuration file format

The configuration file is a file representing a single JSON object with the fields described in Table 4.1.

4.2 Changing the task size

Golemize has significant overhead between Golem tasks: it must import your script, load and unpickle the specific input for the task, and load and unpickle the common data for all tasks. If tasks are short, this is very likely to require more time than computation itself.

As such, Golemize puts more than one input in a Golem task. The inputs are calculated in separate calls to your function, but the setup and teardown overhead only happens once. By default, it uses 10 inputs at a time, but this is likely to be too low for very quick tasks and too high for jobs that have few tasks overall.

Table 4.1: Contents of a Golemizer comfiguration file

name	type	req'd?	notes
serverURL	string	*	Address, including port, of the Golem master server.
serverPassword	string	*	Password for job queue access to the Golem master server.
golemResultRoot	string	*	File path to the root directory for Golem working directories.
lowGolemID	int	*	Lowest-numbered Golem worker ID (usually 1).
highGolemID	int	*	Highest-numbered Golem worker ID (usually the number of workers).
golemStagingRoot	string	*	File path, also visible from the workers, where `golemize.py` can write intermediate files.
golemizeScriptPath	string	*	File path, also visible from the workers, to `golemize.py`.
pythonBin	string		Path to the Python executable on the worker machines.
pickleOut	string		Output path to use for workers. (Leave missing to use Golem working directory.)
taskSize	int		Default number of inputs per task.

As described in the API, the `Golemizer` class allows the taskSize setting to be modified, either by assigning to it directly or by calling into the convenience method `setTaskSize`. While there is a default task size, tuning it for the performance of your computation function is important to the performance of the entire system.

The ideal size is such that each worker gets about three tasks. One task per worker would minimize switching overhead, but providing one more task than planned would provide extremely poor performance when one all workers finish their jobs at similar times, and then that one remaining task is scheduled and all other nodes lay idle while waiting for it to complete. The natural variance in computation time tends to smooth out this effect with more tasks, but beyond 3 tasks per worker, the switching overheads begin to add up significantly.

If you don't know the size of the cluster, or of your input, let the speed of your function guide you. If the total calculation time of a group of inputs takes more than a minute, the distribution overhead is small compared to the calculation, so larger tasks are unnecessary. If you are performing slow, complex calculations, this may set your task size down at 1; if it is a simple operation you need to perform a very large number of times, thousands or even millions may be the appropriate scale.

Experimenting with this setting is probably worthwhile for the curious.

CHAPTER 5

VARIOUS BEHAVIORS

This section discusses various details of the internal operation of `golemize.py`. While these may affect design or usage considerations, they are unlikely to be critical for most uses of the module.

5.1 Three-second polling

Golem neither provides a persistent connection nor pushes a message to notify a client that a job has completed. As such, `golemize.py` must repeatedly ask the Golem server to find out when a job has completed. (This is known as the "are we there yet? are we there yet? are we there yet?" algorithm.)

The polling time is a trade-off between not spamming the server and network with pointless requests, and not waiting idle because the job finished and we haven't asked about it yet.

The polling time is set to three seconds, so goDoIt will pick up results within about three seconds of its job finishing. This decision is arbitrary and may be revisited if this is found to be a stupid polling interval.

5.2 Late exceptions

Exceptions thrown by the calculation function are caught and serialized alongside its other outputs, with a flag set so the loader is aware it has a thrown exception, rather than a result.

goDoIt is intended to behave as if your function was being calculated during iteration of the loop that retrieves results. As such, an exception thrown on the worker is thrown when and only when you iterate to that point in the results, and is thrown by the result generator, with the stack from its original error.

5.3 The result generator is noncaching

`golemize.py` expects to be processing large data sets. As such, it aggressively forgets references as soon as it no longer needs them. Thus, it can handle data sets larger than the amount of RAM available.

- During the input-splitting phase of `goDoIt`, the sequence of inputs is read off from just enough to fill one task, the task is written to disk, and the references to the objects read off the sequence are then forgotten. This allows for large inputs, if a generator (rather than a precalculated sequence) is provided as the primary input.
- During result iteration, result files are loaded only when the previous file has been exhausted and the next result is requested (or it is the first request). This allows for large outputs, if the code using the generator is also not trying to save the output; no references to the values are kept after the generator has moved past the file, so the garbage collector can reclaim it.

There are side effects to this. While this design allows for aggressive memory reclamation, if random access to results is desired, the contents of the generator must be unpacked into a list. The result generator does not know how large the result data set is, so it cannot help such a list be preemptively allocated to its correct size, nor can it provide a measure of how far through the collection the current iteration point is. The generator cannot be reset or reversed.

Since the generator would have to cache results in a list anyway, this just pushes the burden of writing that list onto the user if these features are desired. Aggressively forgetting references allows much larger data sets to be processed.

CHAPTER 6

FAILURE MODES

Like most software packages, `golemize.py` has known conditions under which it will not work. These are fundamental to the design and can't simply be patched around; the ones that could be simply fixed already were.

6.1 Exceptions that are not Exceptions

`golemize.py` can only catch exceptions thrown by the calculation function if those exceptions inherent from class Exception. This is true for all the built-in exceptions, and every exception thrown by every built-in Python library, and every exception thrown from C-based Python extensions. User-written code, however, can `raise` anything.

Due to limitations in the Python language, an exception handler can catch an exception *and use a reference to it* only if its type, or part of its type, is known. While the `catch` statement on its own will catch anything thrown, there is no way to use that statement and keep a reference to the thrown exception. Without a reference, `jumpToTask` has no way to store the exception to rethrow it in the result generator. As such, these "non-exceptions" will simply crash the software on the worker, and its error dump will appear on the master stderr file. `goDoIt` will then fail with an exception noting that it could not find results for every task.

6.2 Scripts not safe for import

The script containing the calculation function will be imported, so if it does not work correctly when imported as a library, it won't work correctly here. This includes checking for __main__. Be careful to ensure that if any initialization is required whether the script is run as a program or not, that initialization happens unconditionally, while script execution behaviors otherwise only happen if the script has been run on its own.

6.3 Nonserializable data

All inputs, all results, and the computation function itself are all written to binary files using Python's `cpickle` library, the standard fast serialization mechanism. If any of these objects cannot be correctly pickled, the platform can't work, and the pile of exceptions from `cpickle` will generally explain the problem in some detail. In these cases, you will generally need to use different classes that do support serialization. Note that `numpy` arrays pickle just fine.

6.4 Missing libraries

To run the calculation function on the nodes, they must import and be able to run the module that function is contained in. If that module itself imports libraries that are not available from the nodes, then the import will fail, and Golem will be unable to run your software.

This is most often caused if your script refers to other code you have written, and golemize.py did not copy the dependencies alongside your script. This is most generally solved by making your script run in place. If you are referring to a Python library that is installed on your local machine, but is not installed on the cluster machines, contact IT for assistance to get that library installed on the computers Golem uses for computation.