Stanford Exploration Project, Report 117, October 23, 2004, pages 175–183

Short Note

Fault tolerant parallel SEPlib

Robert G. Clapp[1]

INTRODUCTION

In the past few years, several papers have been written dealing with SEP's attempts to run on Beowulf style clusters. The initial work involved the use of the Open Multi-Processing (OMP) library (Biondi et al., 1999). As we increased the number of nodes at SEP, we switched to, or added on support for, MPI to many of our programs (Sava and Clapp, 2002). This proved to be a somewhat successful strategy with a few notable drawbacks. First, it required each program to include MPI specific coding. Clapp (2003b) described an effort to minimize the repetitive portions by introducing a library that did MPI-like operations on SEPlib files.

Another problem was that we relied on mounting, or in our case automounting, of disks. Each process would open the history file (and in most cases the binary file) of each input SEPlib file. This posed two problems. First it doesn't scale well to a large number of nodes (100 processes opening up the same file will often fail). Second, automount isn't particularly stable on Linux. Clapp (2004a) was an attempt to address this problem. It worked in a master-slave manner. The master process would read a sepfile and pass along its description and contents to worker nodes. In addition, it introduced the concept of a distributed SEPlib dataset. A SEPlib file could be broken into sections along one of its axes and various parts could sit on different nodes. It provided routines to partition and collect the distributed dataset. Unfortunately this left one glaring problem, node instability. When running a MPI program if a node becomes inoperable the entire job will die either immediately or at the first communication attempt. The solution, to add to each application the ability to 'restart' itself can be extremely challenging, especially in large, global inversion applications (Sava and Biondi, 2003; Clapp, 2003a).

In this paper I describe a library, written in Python, that allows auto-parallelization with a high-level of fault tolerance for almost any SEPlib program. Instead of handling parallelization within a compiled code at the library level, the parallelization is done at the script level which sits on top of the executables. The Python library distributes and collects the datasets, keeps track of what portion of the parallel job are done, and monitors the state of the nodes. The distribution and collection are done through MPI but individual jobs are all serial codes. The

[1]**email:** bob@sep.stanford.eda

code is written using Python's object oriented capabilities so it is easily expandable. A parallel job is described by a series of files and a series of tasks.

PARALLEL JOB

The controlling process for a parallel job is the `SEP.parjob` object. I will begin by going over the objects it inherits properties from. Then I will discuss its initialization and finally go over its execution sequence.

Inheritance

A parallel job is a Python class object that inherits from three other classes. The first is the `SEP.status.sep_status` class. At the most basic level this class simply reads and writes to an ASCII text file. This text file is what keeps track of the progress for the job. Each line of the status file is a ':' separated list. The first item is the text descriptor, the *jobid*, for each task. The second item is the status of the job (`todo,sent,running,finished,collected`). The third is what machine (if any) the job is running on/ran on. The final two are progress indicators for the job (to enable restarting), and the number of times the job has failed. In the course of a parallel job, several different processes will need to read and/or write to the status file. In order to avoid clobbering of the file contents, each process can get an exclusive lock on the status file.

The parallel job class also inherits from a class that handles socket communication `sep_socket.sep_server`. The socket class knows how to find a free socket number, and how to run a socket server that takes actions based on simple string messages. Finally it inherits from a class, `SEP.par_log.jobs`, that stores the stdout and stderr of the various jobs.

Initialization

There are two required parameters and several optional parameters to initialize a parallel job. The first required parameter is (`sect_pars`) a Python dictionary that links the *jobid*mentioned before, with a parameter object `SEP.par.sep_pars`. This class knows how to read, write, add, and return a list of parameters. An example of where a job might need different parameters is if you are creating an output cube that scans over a parameter space. The second required parameter is a dictionary of parallel files that are needed for the job. A dictionary rather than a list is used to enable a parallel job to be used in inversion. When inverting the model and data vectors will change depending on the inversion process. `key` in the dictionary enables an easy way to locate (and possibly change Clapp (2004b)) the parallel file. The `value` in the dictionary is a parallel file object discussed below.

Below is a list of some of the useful optional parameters:

program The serial code that you want to run on each node. This is an optional, rather than required parameter, because it is possible, and in some cases useful, to extend the `SEP.parjob.parjob` class's job creation mechanism.

restart Whether or not we are restarting a job. When restarting the status file will be read instead of created.

global_pars A list of of parameters to run with every job, again a `SEP.par.sep_par` object.

attempts The number of attempts (defaulted to 2) to attempt to start a job (on different nodes) before failing.

restart_com A parameter that needs to be added to the calling sequence when restarting a job.

mfile The machine file to read the available nodes from (in the same status convention as an MPI machine file. Defaults to a file name `mfile`.

njobs The maximum number of jobs to run simultaneously. Defaults to the number of processors described in `mfile`.

verb The level of verbosity.

The initialization creates a machine class object (also inherited from the status class). This object keeps track of the available machines. An obvious extension of this class is discussed in the future work section.

Running

When a parallel job is started it first checks the status of all of the nodes, removing any that aren't functioning properly from its potential list. A second thread is then created.

This new thread handles job distribution. It requests a list of free nodes from the machine class. It then finds the list of the jobs that haven't been completed. It matches the job to the available nodes and requests from the parallel file object a SEPlib tag for a version, or portion, of the parallel file given the requestion section and node. It then builds a command string for the job. In addition to a `rsh` call to the machine and the program name the command string includes the global parameters and section parameters discussed above along with information on how to communicate with the server process. The job and machine's status files are then updated to note what job was sent to what node.

These jobs are started by a series of forked processes. These forked processes will run the job and record in the status file if they ran to completion or failed with an error. These processes will then exit. The job creation thread will run in a loop, checking for available machines and occasionally (every 5 minutes) checking to see if a node has become inaccessible.

The original thread starts an `INET` server socket. It receives communication, in the form of small ASCII strings, from the slave processes. It recognize four different messages: job started, job finished, error in the job, and progress of the job. How the serial code sends these messages is discussed later. If the socket server receives a start message it changes the status of the job from `sent` to `running`. A progress message is added in status file. A finish message changes the job status to finished and marks the node as available. A failure message results in the node status being checked, the job being re-listed as 'todo', and an updating of an internal list to insure that the same job is not sent to the same node. In addition the last 100 lines of stdout and stderr of the failed job are sent to stdout. All messages are also recorded in each parallel file's status file.

When then job creation thread notes that all jobs have run to completion it exits. Finally, when the socket server notes that all jobs have been finished it loops through all of the parallel files. Each output file is combined, and the job is exited.

PARALLEL FILE

Th key building block for a parallel job is the parallel file.

Initialization

A parallel file is initialized by a series of required and optional parameters. The following are all required to initialize a parallel file.

name The name of the sepfile.

tag The tag used by the serial program (e.g. `< , >, tag2=`).

usage How the file is to be used as input (`INPUT`) or output (`OUTPUT`).

isect_dict A python dictionary object that links the job descriptor with the section number it corresponds to.

In addition there several optional parameters available.

type The distribution method for the file. Currently three options are amiable:

- `BLOCK`, different elements of the distributed axis are grouped together;
- `SPREAD`, elements along the axis are distributing in a round robin fashion; or
- `COPY`, the file is shared among all of the sections. If input, the file will be copied to all the nodes. If output, the sectioned files were added together to form the final output. The file will also be shared among all processes running on the node. In order to avoid multiple processes overwriting themselves a simple file locking mechanism is available (see the programming section).

axis The axis the data is distributing along. Required for all distribution methods except `COPY`.

reuse_par A parameter string to include when the file is being reused. This is only applicable in the case of `COPY` distribution type.

Functions

A parallel file is also derived from the status object. The `sent`, is `running`, `finished`, or `failed` messages received and recorded by the parallel job status file are also recorded in the parallel file status file. In addition the parallel job object has the ability to distribute and collect itself. Both of the operations are done through MPI using the external SEPlib function `Partition` and `Collect` programs.

PROGRAM FUNCTIONS

There are two required and three optional function available to the programmer. The required functions go near the beginning and ending of the program. The first, `sep_begin_prog`, tells the server that the program has started successfully. It should be used when it is safe to start another instance of the program on the same node. This latter requirement is important when sharing an output space (`COPY`) on a node. If you are adding to the output space you need insure that the output file has been created by the first instance of the program. The other required function is `sep_end_prog`. This should be used after all of the output files have been completely written to. The server interprets as a sign of completion for the job.

The `sep_progress` function enables effective job restarting. The programmer can use this function to signify a checkpoint in the code. When restarting a job, both the restart flag and the last progress message will be passed as arguments for the restarting job. In the case when you sharing an output space, you should use this function whenever you write to the shared file. The final two functions are when you are sharing an output space. The `sep_open_lock(tag)` locks (if already locked waits for the file to become available) a seplib tag. The `sep_close_lock(tag)` functions frees a tag so another process can safely read or write to it. All of these commands perform no function when not a portion of a parallel job.

FUTURE WORK

There are five additional features that would be useful to add to the parallel infrastructure. These changes need to be made to the dataset collection, the machine selection, and the file distribution options.

Currently datasets are collected only at the completion of the job. It would be useful to be able to concatenate at various time intervals. This would make QCing easier and make the job less susceptible to disks going bad, requiring the rerunning of portions of the job. The problem with implementing this feature is when the output file is of the `COPY` type. The file is being continually updated by processes running on the node. In order to get an accurate picture of what is collected all writing to the node would have to bet frozen and/or insure that no processes is running on the node. The second change is in the actual methodology of the collection. Currently a binary tree sum is done using the `MPI_collect` routine. For large datasets this approach quickly swamps the network switch. A more efficient collection method should be implemented.

Another feature would be to expand on the current `COPY`, `BLOCK`, and `SPREAD` methods. It would be useful to be able to distribute over multiple axes, and to have the distributed portions overlap (patching).

Finally, a couple changes to the way the machines are chosen needs to be made. Instead of the list of available machines being read from a file it would be useful to have it read from a global server. n this way you could have dynamic control over the number of nodes a job was running on. In addition in some special cases it is necessary to guarantee that jobs run on specific nodes. Wave equation migration velocity analysis (Sava and Biondi, 2003) is an example of this. The downward continued wave-field is pre-stored on the nodes. You must be able to insure that each frequency is sent to the node that contains the wave-filed at that frequency.

The object oriented way that the library was implemented makes these changes relatively easy. For example, to insure that a specific job is spent to a specific node would involve inheriting the parallel job class, and overriding the routine that matches jobs with available machines.

CONCLUSION

A fault tollerant, parallel job enviornment is created within python. Serial codes can be in parallel with minimal changes. The object orriend nature of the library makes it easily extendable to solve almost any coarse grained problem.

REFERENCES

Biondi, B., Clapp, R. G., and Rickett, J., 1999, Testing linux multiprocessors for seismic imaging: SEP–**102**, 235–248.

Clapp, M. L., 2003a, Velocity sensitivity of subsalt imaging through regularized inversion: SEP–**114**, 57–66.

Clapp, R. G., 2003b, MPI in SEPlib: SEP–**113**, 491–500.

Clapp, R. G., 2004a, Parallel datasets in seplib: SEP–**115**, 479–488.

Clapp, R. G., 2004b, A python solver for out of core, fault tolerant inversion: SEP–**117**.

Sava, P., and Biondi, B., 2003, Wave-equation mva: Born rytov and beyond: SEP–**114**, 83–94.

Sava, P., and Clapp, R. G., 2002, Wei: a Wave-Equation Imaging library: SEP–**111**, 383–395.

APPENDIX A

The following is a simple example parallel job. In this case a file is split along the third axis and the SEPlib program `Scale` is run on each section, then the files combined to form the final output.

```
njobs =4          #the number of jobs we are going to break the problem into
isect_dict={}     #an empty dictionary object relating the section and jobid
par_list={}       #an empty dictionary object relating parameters and jobid
for i in range(njobs):           #loop over the jobs
   isect_dict[str(i)]=ia         #form the jobid:isect dictionary ('1':1)
   par_list[str(i)]=SEP.par.sep_pars(name=str(i)) #form an empty parameter list

file={}    #the files dictionary
#create the input: first command line is the name of the file, referenced with<
file["input"]=SEP.parfile.parfile(name=sys.argv[1],
  tag="<",isect_dict=isect_dict, usage="INPUT",axis=3,verb=verb)
#create the output parallel file object
file["output"]=SEP.parfile.parfile(name=sys.argv[2],tag=">",
isect_dict=isect_dict, usage="OUTPUT",axis=3,verb=verb)
#create the parallel job
test1=SEP.parjob.parjob(files=file,sect_pars=par_list,program="Scale",
 log_level=2,verb=verb,njobs=njobs)
#run the job
test1.run_job()
#clean: remove all the temporary and status files
test1.clean()
```