Stanford Exploration Project, Report 115, May 22, 2004, pages 479–??

Short Note

Parallel datasets in SEPlib

Robert G. Clapp[1]

Cluster computing, with relatively inexpensive computational units, looks to be the most cost effective high performance computing option for at least the next five years. Unfortunately, cluster computing is not the most convenient processing environment for the type of problems routinely performed in seismic exploration.

Many of the tasks we want to perform require both large memory and massive disk space. These requirements make grid based computing (Foster and Kesselman, 1998) generally impractical. Storing and accessing these large datasets is also problematic. The Parallel Virtual File System (PVFS) (Ross et al., 2004) is an attempt to create a virtual file system across several machines. It is still relatively new, and doesn't allow good user control of data locality (e.g. you often know you want to store frequency 'x' on node one because that is where you plan to process it).

When writing an application for a cluster you often face difficult additional challenges. Problems specific to coding in SEPlib, but which are indicative of more wide-spread problems include: each process may not be able to read a shared a history file; each process needs a unique tag descriptor when writing out a temporary file; all shared binary files need to be read locally and then distributed (through some mechanism such as MPI (Snir et al., 1995)); and writing effective check-pointing in a parallel code can be extremely cumbersome. These problems significantly increase the complexity or writing and debugging applications.

In this paper I discuss an extension of the basic SEP data description to help solve the above problems. It extends the definition of a SEPlib dataset to include one that is stored on multiple machines. It allows the same code base to be used for both serial and parallel applications. All that changes is which libraries you link with. Parallel datasets can be easily created and accessed, greatly simplifying coding and debugging in a cluster environment. In this paper I describe how the library works, and provide several examples of its use.

DESIGN

The extension of SEPlib to handle parallel datasets has many unintended similarities to the parallel data capability of HDF (NCSA, 2004). A parallel dataset is a composition of several datasets with a master file describing the relationship between the various parts. All programming with the parallel dataset library is done through the `superset` library (Clapp, 2003). To

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

handle parallel datasets with minimal changes to coding style requires a more abstract definition for IO than simple SEPlib or SEP3d allows. The library has both C and Fortran90 interfaces which allows almost any SEPlib code to be changed quickly. In this section I will describe how to interact how the library interacts with the various sub-datasets.

Files

Similar to SEP3D (Biondi et al., 1996), the master file is still the standard SEP history file. The difference is that the history file will not have a pointer to the dataset (`in=`) and it will have an additional pointer to a text file (Distribution Format File (DFF)) describing the distribution pattern (`dff=`).

The distribution format file is made up of several parameters:

nsect The number of sections the dataset is broken up into.

axis The axis along which the data is distributed.

naxis_convert The length of the axis that the data is distributed along.

pattern The distribution pattern along the distributed axis. Right now two distribution patterns are defined, `BLOCK` and `SPREAD`. In the first option sequential elements along the first axis will belong to the same section. In the second option a given section will contain elements $isect + k * nsect$ where $isect$ is the section number, $nsect$ is the number of sections, and k is the number of axis elements the section owns. One obvious future extension to the library is to handle more arbitrary distribution patterns.

tag_sect*i* The tag associated with section number *i*.

mach_sect*i* The machine in which section *i* was written.

The final set of files are the various `tag_sect`. Each `tag_sect` is a valid SEP3D dataset.

Initialization

There are three mechanisms to initialize a new distributed dataset. A distributed dataset can be created by a call to the library. The call defines which axis to distribute, how many sections to create, and what distribution pattern to use. The library then determines the number of threads it is running on and then distributes the sections to the various threads in a round-robin fashion. Normally you would think of having a single section for each thread, but this is not required, and in certain situations might not be desirable.

A distributed dataset can be read from a file. When initializing a distributed dataset from a tag, data locality tries to be preserved. The various sections are assigned to threads on the same machine as much as possible without sacrificing load balance. For example, if a dataset

has been broken up into six sections, three on machine A and three on machine B, and you then start a job on machine A, B, and C, A and B will each be assigned two sections that were written locally, while C be assigned one section from both the original A and B group. Whenever a dataset is initialized from a tag, the tag is only read by the master node. Its contents are then sent to all slave processes. This keeps every process from having to have access, and to simultaneously read, a single file. When scaling to a large number of processors, this ability can be useful.

The final method to create a distributed dataset is from another distributed dataset. If you are running on multiple threads and you initialize a new dataset from an already distributed dataset, the new dataset will inherit the original dataset's distribution pattern.

I/O

The library operates in three different modes when reading and writing a dataset. When reading a non-distributed dataset in a parallel environment, the dataset will be by default read by the master thread and sent to all slave threads. You can also define the read as being exclusively local (imagine a local temporary file). When reading a distributed dataset, the read will just return the portion of the given read request that it locally owns. This requires you always to check the amount of data that is returned. In the next section I will provide an example.

When writing a distributed dataset, the library will by default assume that you are writing only the portion of the given window that it actually owns. When writing a non-distributed dataset you can again specify that a write is exclusively local, but otherwise only the master thread will write data, and it is the programmer's job to make sure they have collected the dataset.

Distributing and collecting

Distributing a dataset is rather trivial. You simply have the master thread read the dataset locally and then use the distribute dataset routine to partition the dataset to the various threads. Collecting operates in a similar manner. The various threads read the data locally and then use a collect routine to combine the various subsections. The library also offers the ability to compress dataset along an axis. An example of this would be migration, where each thread will have its own image and the master thread wants the sum of the various portions.

Parameter handling

One of the biggest problems with running on cheap hardware is the failure rate is high. Writing migration code that is able to figure out where it died and continue is challenging. When the migration is part of a larger inversion problem the tasking becomes even more difficult. One of the goals of the library is to make check-pointing easier. Any thread can write a status parameter to a distributed tag. This status parameter is written to its local sections rather than

the global tag, so clobbering of the text file isn't an issue. Restarting becomes a much simpler matter. You can request the status parameter from each section with a single call. Figuring out what portion of job has finished, and what portion is remaining becomes a trivial matter.

Global parameters

The library offers two options that are available to any distributed dataset. The first option, `master_data`, allows you to switch back and forth between a master-slave and master among equals programming paradigm. If `out.master_data=1`, the default, the master thread will own section(s) of the distributed tag `out`, otherwise it will not. Running in a true master-slave mode can be useful when your data is for a file server which you do not want to do computations on.

The second option is to guarantee that each section of dataset will have a unique name. By default all sections are given a name based on the output tag name (when accessible[2]), otherwise a random, unique name is given. A unique name can be guaranteed by specifying `out.unique=1`, where `out` is the tag, on the command line. A related option is function call, `sep3d_make_local_name` which will create a tag on a local machine with an unique name.

Extendability

The library was written with the assumption that current parallel programming paradigms are likely to change. All of the MPI specific code is limited to two files. Adding support for Parallel Virtual Machine (PVM) or the next 'latest and greatest' parallel programming library simply involves defining how to transfer data between processes and how to synchronize their flow.

WRITING APPLICATIONS

The best way to learn how to write code is to look at examples. Rather than just make list of function calls, I will take portions from several different codes.

Serial

The first example is from the SEPlib program `Nmo3d`. In SEPlib there is a module that helps initialize the parallel environment. Our only `ifdef` will be in whether to include the MPI version of the parallel environment or a dummy serial version. We also need to initialize our parallel environment. The optional arguments signify the input and output to the program. If provided to the `sep_par_start` routine, `intag` will be set to "in" for a serial run and "intag" for a parallel run. The variable `outtag` will be set to "out" or "outtag" and will be initialized with a copy of the `intag` history file.

[2]There is no standard way in unix to find the name of a file when it isn't written the local directory

```
#ifdef SEP_MPI
  use sep_par_mpi
#else
  use sep_par
#endif
  call sep_par_start(intag=intag,outtag=outtag)
```

Initializing the output dataset is unaffected. If we are running in a parallel environment and the input is sectioned, the output tag will be sectioned. Otherwise a standard SEPlib file will be created.

```
  call init_sep3d(in,out,"OUTPUT",ctag=outtag)  !Initialize SEP
```

It is often useful to know the thread number for the current process. The below code section bases its decision on how often to print progress on whether it is the master or slave process.

```
  if(sep_thread_num()==0) then
    call from_param("pct_master",pct_print,2.)
  else
    call from_param("pct_slave",pct_print,10.)
  end if
```

For NMO our data can be gridded in an arbitrary style. We simply need to loop through all of the data. The `init_loop_calc` function figures out looping parameters based on the dimensionality of the data `in%ndims`, the dimensions of the data `in%n`, and the maximum amount of data we want to read `ntr*in%n(1)*in%n(2)`. The identifier `"MAIN"` just gives a name for the current loop. The `do_sep_loop` will increment windowing parameters `fwind` and `nwind` until we have looped through entire dataset, at which point the return value will change.

```
  if(0/=init_loop_calc(in%ndims,in%n,"MAIN",ntr*in%n(1)*in%n(2))) &
     call seperr("trouble initing loop")
   do while(0==do_sep_loop("MAIN",nwind,fwind))
```

To read the data (whether it is a regular or irregular cube) we first request the number of traces that fall in our current window. In this case, the library will return `nh`, the number of traces that the current thread locally owns in the given window.

```
   call sep3d_grab_headers(intag,in,nh,fwind=fwind(2:),nwind=nwind(2:))
```

If the thread owns any of data in the window, read it.

```
   if (nh.ne.0) then
     if (.not. sep3d_read_data(intag,in,input(:,:nh))) then
          call erexit("trouble reading data")
       end if
```

For a regular dataset, which elements of the current window are owned by the current thread is important. We can request to know the coordinates of the traces. The returned 2-D array is of dimensions ($ndim-1$ by nh), where $ndim$ is the dimensions of the data and nh is the number of traces in the current block.

```
if(.not. sep3d_grab_coords(in,coords)) &
          call seperr("trouble grabbing coords")
```

Writing is again automatic. We will automatically write just the local portion of our data.

```
if(.not. sep3d_write_data(outtag,out,output,nwind=nwind,fwind=fwind)) then
   call seperr("trouble writing out ")
end if
```

The only remaining step we need to do is to make sure that the total number of traces is correctly calculated and make any calls needed to end the current parallel environment. The library keeps track of how many traces it has written to a given file. The `sep3d_update_ntraces` call tells the master thread how many traces each slave thread has written. The total number is then written to the output history (and possibly header format file).

```
if(.not. sep3d_update_ntraces(out)) call seperr("trouble updating ntraces")
  call sep3d_rite_num_traces(outtag,out)
call sep_par_stop()
```

Distributing a dataset

The second example is taken from the SEPlib program `Transf`. It does a Fourier transform of a dataset and then distributes the data along the frequency axis. Again, only the portions relevant (and different from the previous example) will be shown.

In this case we want to know how many threads we are running on.

```
nmpi=sep_num_thread()
```

By default, when converting to frequency `inv=.false.`, we are going to create a dataset with $nmpi$ sections, but we will allow the user to override this option. We are going to section the dataset along the last axis ($ndimc$) which in this case is the frequency axis.

```
nsect=1; if(.not. inv) nsect=nmpi
call from_param("nsect",nsect,nsect)
if(nsect>1) then
  if(.not. inv)  then
    if(.not. sep3d_section_tag(output,ndimc,"BLOCK",nsect)) &
       call seperr("trouble sectioning tag")
```

Our input and our output could involve a sectioned dataset. Given our current window parameters (nc,fc,jc for output and nr,fr,and jr for our input), we need to know the size of the data with which we will be working. The `sep_local_buffer_size` gives the dimensions (number of samples by number of traces) of the local buffer size given the current window for a given dataset.

```
if(.not. sep3d_local_buffer_size(output,n_c,nc,fc,jc) .or. &
      .not. sep3d_local_buffer_size(input,n_r,nr,fr,jr)) then
     call seperr("trouble getting local buffer sizes")
   end if
```

Once we have transformed the data, we need to pass it to the appropriate thread. We first describe the window that we want to pass `sep3d_set_window` and then distribute the data.

```
call sep3d_set_window(space,nwind=nc,fwind=fc)
    if(.not. sep3d_transfer_data(input,space,blockc_in,blockc_out))&
      call seperr("trouble transfering data")
```

Everything else that is different from how you would conventionally program is the same as the first example.

Restarting and combining

The following examples are taken from the SEPlib WEI library(Sava and Clapp, 2002). They demonstrate how to read a dataset, recognize which elements of the dataset are locally owned, and write status parameter and how to combine a dataset.

This first code portion is executed every time a frequency has been fully migrated. The frequency number currently being processed i_w are looped through. A variable named `ifreq_doneX`, where 'X' is the frequency number is created. Then that frequency is set to to 1 by the `sep3d_set_sect_param`. In this case the image sep file (the sep3d structure `rsep`) is potentially a distributed dataset along an aritificial axis corresponding to the number of processes that are computing the image. The `sep3d_set_sect_param` call will automatically write this parameter to its local portion of the image.

```
do i=1,size(i_w)
   par="ifreq_done"
   call parcat(par,i_w(i))
  if(adj) then
    if(.not. sep3d_set_sect_param(rsep,par,1)) then
      write(0,*) "trouble writing sect_params"
      return
    end if
```

The second example is from the portion of the code that checks for a restart request. Here, all of the threads are looped over. For each frequency, we check all of the sections of the image tag to see if any of the threads have processed this frequency.

```
do ifreq=1,wsep%n(6) !loop over frequencies
  par="ifreq_done"
  call parcat(par,ifreq)
  iw=-1;
  if(.not. sep3d_grab_sect_param(sep,par,iw)) then
    write(0,*) "trouble grabbing section parameters"
    return
  end if
  if(any(iw==1)) then
```

Oftentimes we need to make a decision whether to proceed based on whether we own a specific element of the axis we spread along. This code fragment is marking whether or not it owns a given frequency.

```
do i=1,wsep%n(6)
  iw_own(i)=sep3d_own(dsep,i)
end do
```

In the case of migration, we need to combine all the local images into a global image. This code fragment adds all of the local data (`bigc`) from the distributed image (`big_sep`), into the standard SEP dataset (`small_sep`) buffer `bigc2`.

```
      if(.not. sep3d_compress_data(big_sep,small_sep,bigc,bigc2)) then
             write(0,*) "trouble combining data ", fwind
             return
         end if
```

FUTURE WORK

There several additional features that could be added to the current libraries' capabilities. As mentioned earlier an arbitrary description about how the data is distributed along an axis would be useful. This would allow for better load balancing, redistributing of tasks on a restart, and for operations that perform some type of windowing.

The most obvious place for work is to add recognition of distributed datasets to more SEPlib programs. Currently only the generic SEPlib utilities `Window3d` and `In3d` are aware of distributed datasets. Adding awareness to program like `Headermath` would be helpful. More application programs could/should be converted. Currently `Transf`, `Nmo3d`, `Phase`, `CAM`, `Rtm2D`, and `Fdmod` are distributed dataset aware. Almost all of SEP's current application programs could benefit from conversion.

CONCLUSIONS

Developing code for a parallel environment requires significant, sometimes non-obvious additional coding overhead. By expanding the concept of a SEPlib dataset to include a dataset that is spread over several machines, much of the difficulty associated with cluster computing can be avoided by the programmer. The included examples show how with relatively minor changes in coding style a parallel code can be created virtually free.

REFERENCES

Biondi, B., Clapp, R., and Crawley, S., 1996, Seplib90: Seplib for 3-D prestack data: SEP–**92**, 343–364.

Clapp, R. G., 2003, SEPlib programming and irregular data: SEP–**113**, 479–490.

Foster, I., and Kesselman, C., 1998, The grid : Blueprint for a new computing infrastructure: Morgan Kaufmann.

NCSA, 2004, The ncsa hdf home page: `http://hdf.ncsa.uiuc.edu/`.

Ross, R., Ligon, W., Carns, P., Miller, N., and Latham, R., 2004, Parallel virtual file system: `http://www.pvfs.org/pvfs2/`.

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

Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., and Dongarra, J., 1995, Mpi: The complete reference: MPI Press.