Short Note

MPI in SEPlib

Robert G. Clapp[1]

In the last 5 years, a dramatic shift has taken place in the way high-performance computation is done in the oil industry (Bednar et al., 1999; Clapp and Sava, 2002) and many other fields that have heavy computation requirements. No longer is a single computer with massive shared memory and multiple processors the economical choice. Instead numerous inexpensive PC based machines are clustered together for computation. This change in hardware paradigms requires a corresponding change in programing paradigms. No longer are shared memory programing models such as Open MP is sufficient. Instead a more distributed model such as Parallel Virtual Machine (PVM) or more commonly, Message Passing Interface (MPI) must be used.

A second challenge with this change in paradigm is related to the type of problems that we deal with in the oil industry. The volume of data makes network transfer a non-trivial portion of the processing flow and is the dominant element in some (Mosher, 1991). In this paper, I describe the routines in SEPlib to facilitate MPI processing and discuss some future possible directions.

MPI WITH SEPLIB FILES

For many applications, a very few routines are all the MPI that is needed. The basic idea of these routines is to make a local version of global files. These files are transfered to and from the master process through some simple routines. These routines can be broken into three categories: initialization, distribution, and collection. All of the routines are written in C with a Fortran interface (e.g. floats become reals, ints become integers).

Initializers/Destructors

mpi_sep_send_args(int nproc, int max_mem, int verb) - Transfer parameters from the master node to all the slave nodes. Only issued by the master node.

nproc - Number of processes in the MPI job.

max_mem - Maximum memory to use when transferring data.

verb - Whether or not to be verbose when transferring.

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

mpi_sep_receive_args () - Receive arguments from the master node. MPI standard does not require command line arguments to be passed to slave processes. This is SEPlib's solution to the problem.

mpi_sep_clean () - Deallocate the memory used for transferring.

Distribution

All of the values are in terms of 4 byte words and assume that the file has been moved (`sseek`) to the correct position. Each function returns an integer, `0` if successful.

mpi_sep_tag_bcast(int i_from,char *t_from, int i, char *t_to, int bs, int nb) - This routine makes local copies of a given tag.

i_from - The master node containing the file.

t_from - The tag of the sepfile on the master node.

i - The thread number of the calling process.

t_to - The tag for the local version of the file.

nb,bs - The size of the file $nb * bs$. Broken into parts to allow larger than 2GB.

mpi_sep_tag_distribute(int i_from,char *t_from, int i, char *t_to, int bs, int nb, int *s_to) - Distribute a tag to the various nodes.

i_from - The master node containing the file.

t_from - The tag of the sepfile on the master node.

i - The thread number of the calling process.

t_to - The tag for the local version of the file.

bs - The blocksize of the various dataset portions.

nb - The number of blocks in the dataset.

s_to - An array of size `nb` telling what node should have each portion of the dataset.

Collection

All of the values are in terms of 4 byte words and assume that the file has been positioned (`sseek`) to the correct position. Each function returns an integer, value 0, if successful.

mpi_sep_tag_sum(int i_from,char *t_from, int i_to, char *t_to, int bs, int nb, int add) - Sum the results of the various local versions into one file.

i_from - The local thread number.

t_from - The local tag name.

i_to - The thread where the image will be collected.

t_to - The tag for the global file.

bs,nb - The size of the file to combine $nb * bs$.

add - Whether (`1`) or not (`0`) to add the result to its current contents of the output tag.

mpi_sep_tag_combine(int i_from,char *t_from, int i_to, char *t_to, int bs, int nb, int *s_from,int add) - Combine the local files into a single file.

i_from - The local thread number.

t_from - The local tag name.

i_to - The thread where the image will be collected.

t_to - The tag for the global file.

bs - The size of the blocks to combine.

nb - The number of blocks.

s_from - An array of size `nb` listing which process has component of the global file.

add - Whether (`1`) or not (`0`) to add the result to its current values.

Appendix A contains a simple program illustrating how to use the library.

AUTOMATIC PARALLELIZATION

The library routines described earlier are effective in many situations. They become difficult to use when the distribution requirements become complex and inefficient when IO transfer times dominate processing times. Another set of routines is available when facing a task for which the first set of routines does not provide an effective solution.

The idea of this set of routines is twofold. First, a tag is distributed along some axis in blocks. This distribution can take two forms: sequential (`SEQUENTIAL`) and combined (`COMBINED`). Figure 1 illustrates the two distribution patterns. The second principle is that there will be three thread types. The master thread will distribute and collect data. Each slave processor will then run two threads simultaneously. The first will simply be concerned with receiving and sending IO to the master node. The second will be processing the data. IO threads will notify its processing thread whenever it has finished reading in a block. The processing thread will then read in its local input, process the data and notify the IO thread that it is safe to transfer back the information to the master node. This will generally reduce the processing time from

$$t_{tot} = 2 * t_i + t_o + t_{proc} \tag{1}$$

to

$$t_{tot} = min(t_i t_o, t_{proc}) + t_{edge} \tag{2}$$

where t_{tot} is the total processing time, t_i is the time to send the input, t_o is the time to send the output, t_{edge} is the time to transfer first input and last output block, and t_{proc} is the total

processing time. This methodology is most effective if the master thread can handle higher bandwidth than the slave threads.

The routines to do MPI in this manner can be broken into four portions. First, constructors and destructors, which begin and end the MPI processing and describe how the data is distributed. The second batch of functions provide information to the programmer about what a given thread will be doing. The final two groups are specific to processing and IO threads.

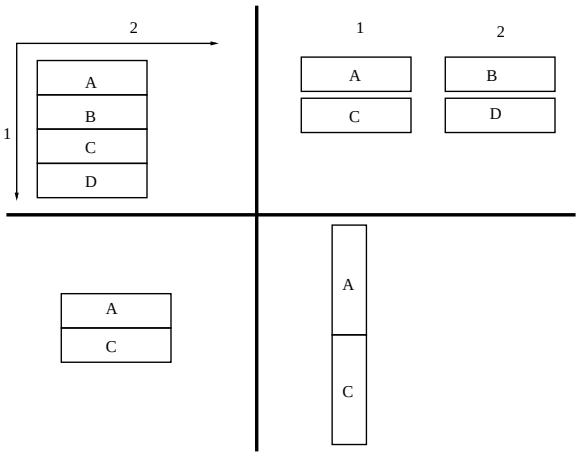

Figure 1: The different distribution patterns. The top-left panel shows a dataset distributed into 4 parts along the first axis (you would normally never distribute along the first axis but the argument holds true for any axis distribution < ndim) . In the top-right panel note that sections A and C go to processor 1 while B and D go to processor 2. The bottom two panels show the two possible distribution patterns. In the combined distribution the blocks retain their normal relationship. In the sequential distribution the processed blocks are written sequentially. bob4-dis [NR]

Constructors and destructors

sep_mpi_auto_io_init() - Initialize the automatic IO routines for the program.

sep_mpi_finish() - Finish all of the MPI processes and clean up intermediate files.

sep_mpi_distrib_in_tag(char *tag_in, char *io_tag, int iaxis, int nblock,char *distrib_type)
- Distribute an input tag.

tag_in - The SEP3D tag name for the dataset.

io_tag - The history file associated with the dataset.

iaxis - The axis to distribute along.

nblock - The block size (in iaxis elements) for each distributed portion.

distrib_type - What type of distribution pattern to use for the given tag.

sep_mpi_distrib_out_tag(char *tag_in, char *io_tag, int iaxis, int nblock,char *distrib_type)
-Distribute an output tag.

tag_in - The SEP3D tag name for the dataset.

io_tag - The history file associated with the dataset.

iaxis - The axis to distribute along.

nblock - The block size (in iaxis elements) for each distributed portion.

distrib_type - What type of distribution pattern to use for the given tag.

Process information

nproc=sep_mpi_proc_type - Return the type of process for the tag.

nproc=0 - The thread is the master for the process.

+iproc - The thread is for processing.

-proc - The thread is for IO (-1 * thread number is returned).

nport=sep_mpi_thread_portion(char *tag, int ithread) - Portion that the given thread must process.

tag - The tag to get the information about.

ithread - The thread number returned by `sep_mpi_proc_type`.

nport - The total number of elements in the distributed axis that this thread needs to process.

Processing thread routines

sep_mpi_proc_begin(char *intag,char *outtag) - Get the local tag names for the input and output.

intag - The local tag name for the input.

outtag - The local tag name for the output.

sep_mpi_begin_io(char *tag,int nblock) - Return when IO can be begun on a given block.

tag - The local tag name.

nblock - The block that I need to have transfered before beginning IO.

sep_mpi_update_block(char *tag,int nblock,int input) - Record that the current block has finished processing.

tag - The local tag name.

nblock - The block that has been finished.

input - Whether this is as input [1] or output tag.

Transferring thread routines

sep_mpi_distrib_all(char *tag) - Distribute a tag to slave processes.

tag - Tag to distribute from the master node.

sep_mpi_collect_all(char *tag) - Collect a tag from the slave processes.

tag - Tag to collect on the master thread.

Appendix B contains an example program using this IO framework.

FUTURE WORK

There are several directions to extend this work. First, there is no reason to limit the sending and receiving of data to regular datasets. The library is based on the `superset` library (Clapp, 2003) so extending it to work on irregular data is straight-forward. Second, many operations are a series of parallel operations. An intelligent extension would be to introduce the concept of a dataset spread over many machines. The user would choose whether or not to perform the collection operation or leave local versions. Most of this mechanism is already available in SEPlib (Clapp, 1999). Another extension is to incorporate some of the ideas mentioned in Clapp and Sava (2002). Specifically, with numerous users and diverse needs some type of dynamic allocation and sharing ability is needed. This would require each block to be considered a task and a check would have to be performed before executing the next section.

CONCLUSIONS

In this paper, I summarize some of the routines available in SEPlib for MPI processing. The routines break into two categories. The first is a collection of routines that simulate MPI operations of SEPlib files. The second set allows for some degree of automatic parallelization.

REFERENCES

Bednar, J., Bednar, C., Neale, G., and Thorson, J. Prestack imaging, modeling, and multiple suppression on a Beowulf cluster:, 1999.

Clapp, R. G., and Sava, P., 2002, Cluster building and running at SEP: SEP–**111**, 403–411.

Clapp, R. G., 1999, Additions to seplib: SEP–**102**, 201–218.

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

Mosher, C., 1991, A benchmark suite for parallel seismic processing: A benchmark suite for parallel seismic processing:, Soc. of Expl. Geophys., 61st Ann. Internat. Mtg, 360–362.

APPENDIX A

```
program reduce_it{
use reshape_mod
use sep
use mpi_sep
integer :: i,ndo,mpi_status,ithread,nthread
integer :: ierr,my_status,i,nlocal
integer, allocatable  :: isend(:)
type(sep3d) :: input,output
real, pointer :: array(:,:)
character(len=256) :: intag,outtag
character(len=256) :: localin,localout
integer,external :: mpi_sep_tag_distribute,mpi_sep_tag_collect
call MPI_INIT(ierr) !INITIALIZE MPI
call set_no_putch() !DON'T WRITE TO THE HISTORY FILE
call sep_init(  )   !SETUP SEP IO
!INITIALIZE AUTO IO
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nthread,ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, ithread,ierr)
  if(ithread==0) then  !IF MASTER THREAD
     call from_param("verb",verb,.false.) !BE VERBOSE
     call from_param("mem",mem,10)        !USE 10 MB
     iverb=0;if(verb) iverb=1
     call MPI_SEP_SEND_ARGS(nthread,mem,iverb)  !SEND ARGUMENTS
  else   !SLAVE THREAD
     call MPI_SEP_RECEIVE_ARGS()       !RECEIVE THREAD
  end if
intag="intag";outtag="outtag" !GLOBAL INPUT/OUTPUT TAGS
write(localin,"(a,i2)") "intag",i !CREATE LOCAL INPUT TAG
write(localout,"(a,i2)") "outtag",i !CREATE LOCAL OUTPUT TAG
call init_sep3d(intag,input,"INPUT") !READ IN DATASET PARAMTERS
!MASTER TAG WRITES OUT TO OUTTAG
if(ithread==0){
  call set_yes_putch();
  call set_alternate_putch(outtag)
}
!FIGURE OUT WHERE TO SEND STUFF
allocate(isend(input%n(3)))
ito=1; nlocal=0
do i=1,size(isend){
  isend(i)=ito;
  if(ito==i) nlocal+=1
  if(inext==n) ito=1
```

```
  else ito+=1
}
!DISTRIBUTE THE INPUT
call auxout(localin)
if(0/=mpi_sep_tag_distribute(0,intag,ithread,localin,input%n(1)*input%n(2),&
  input%n(3), isend)) call seperr("trouble distributing data ");
call auxclose(localin); call auxin(localin)
allocate(array(input%n(1),input%n(2)))  !ALLOCATE DATA
do i=1,nlocal{
  call sreed(localin,array,size(array)*4)  !READ IN DATA FROM LOCAL TAG
  array=array*4                            !SCALE DAYA
  call srite(localout,array,size(array)*4) !WRITE OUT DATA TO LOCAL TAG
}
!COLLECT THE OUTPUT
call auxclose(localout); call auxin(localout)
if(0/=mpi_sep_tag_collect(0,outtag,ithread,localout,input%n(1)*input%n(2),&
  input%n(3), isend)) call seperr("trouble collecting output ");
call MPI_FINALIZE(ierr) !CLEAN UP
}
```

APPENDIX B

```
program reduce_it{
use reshape_mod
use sep
use mpi_sep
integer :: n(3),ns(3),no(4)
integer :: i,ndo,mpi_status
integer :: ierr,my_status
real :: o(3),os(3),d(3),ds(3),oo(4),do(4)
type(sep3d) :: input,output
real, pointer :: array(:,:)
character(len=256) :: intag,outtag
integer,external :: sep_mpi_proc_type,sep_mpi_auto_io_init
integer,external :: sep_mpi_distrib_all,sep_mpi_collect_all
integer,external :: sep_mpi_distrib_in_tag,sep_mpi_distrib_out_tag
integer,external :: sep_mpi_in_out_tags,sep_mpi_thread_portion
call MPI_INIT(ierr) !INITIALIZE MPI
call set_no_putch() !DON'T WRITE TO THE HISTORY FILE
call sep_init("")   !SETUP SEP IO
!INITIALIZE AUTO IO
if(0/=sep_mpi_auto_io_init()) call seperr("trouble initializing MPI IO ")
my_status=sep_mpi_proc_type() !GET THREAD NUMBER/TYPE
intag="intag";outtag="outtag" !GLOBAL INPUT/OUTPUT TAGS
call init_sep3d(intag,input,"INPUT") !READ IN DATASET PARAMTERS
!MASTER TAG WRITES OUT TO OUTTAG
if(my_status==0){
  call set_yes_putch();
  call set_alternate_putch(outtag)
}
!DISTRIBUTE THE INPUT ALONG THE THIRD AXIS SIZE 1
if(0/=sep_mpi_distrib_in_tag(input%tag,intag,3,1,"SEQUENTIAL"))&
  call seperr("trouble distributing in tag")
!DISTRIBUTE THE OUTPUT ALONG THE THIRD AXIS SIZE 1
if(0/=sep_mpi_distrib_out_tag(output%tag,outtag,3,1,"SEQUENTIAL"))&
  call seperr("trouble distributing out tag")
if( my_status <=0){  !IO/MASTER TAG
  if(0/=sep_mpi_distrib_all(input%tag))   !DISTRIBUTE DATA
    call seperr("trouble distributing input")
  if(0/=sep_mpi_collect_all(output%tag))& !COLLECT DATA
    call seperr("trouble collecting output")
}else{  !PROCESSING TAG
  !GET THE AMOUNT OF AXIS 3 I AM PROCESSING
  ndo=sep_mpi_thread_portion(intag,abs(my_status)-1)
```

```
  if(ndo<0) call seperr("trouble grabbing my_thread_portion")
  call sep_mpi_proc_begin(intag,outtag) !GET THE LOCAL TAG NAMES
  allocate(array(input%n(1),input%n(2)))  !ALLOCATE DATA
  do i=1,ndo{
    call sreed(intag,array,size(array)*4)  !READ IN DATA FROM LOCAL TAG
    array=array*4                          !SCALE DAYA
    call srite(outtag,array,size(array)*4) !WRITE OUT DATA TO LOCAL TAG
    call sep_mpi_update_block(outtag,i-1,0)!MARK BLOCK COMPLETED
  }
}
call sep_mpi_finish() !CLEAN UP
}
```