Organizing MPI parallel Simulations

Harald Klimach

harald.klimach@uni-siegen.de

Thanks to Rolf Rabenseifner

28th WSSP 9.-10.10. Stuttgart

Interacting with the User in MPI-Parallel Applications

- Getting configuration parameters from the user
- Informing the user about progress and what is being done (logging)
- Reacting to errors that the application detects, and reporting them to the user

Fundamental Common Tasks

- Implemented in a library: SOIL
  - **S**imulation **O**rganization and **I**nfrastructure **L**ibrary
- Utilization of Fypp for pre-processing
- Utilization of waf for configuration and building

Interacting with the User in MPI-Parallel Applications

- **Getting configuration parameters from the user**
- Informing the user about progress and what is being done (logging)
- Reacting to errors that the application detects, and reporting them to the user

User Input: Lua Scripts

- We use Lua scripts as input
  - Aotus
  - User defines required parameters as variables in the script
  - Allows usage of arithmetic and loops in definitions

- Lua scripts might “require“ other script files
  - Lua will search for “required“ files in various places
  - Will stress Meta-Data server, if done by all processes

Reading the Input Just Once and Then Broadcasting

- Avoiding massive overload of filesystem:
- Root process executes the script and loads all „required“ files into memory
  - uses function overwrites to keep track of required code chunks
- Broadcast all the Lua code to remaining processes
- Non-root process can execute the Lua script without accessing the filesystem at all

Lua Require on Root Process

- Before reading the Lua script from file some Lua code is executed to replace the require command
- This new require keeps track of „required“ files in a table, takes care of nested requires
- Table then contains module name and code of required file

Lua Require on Other Processes

- On all other processes require is replaced by a function that does not look for files but in a table with module names and code instead

Broadcasting Configuration

- Opening Lua script on root not only executes it in the root process but also returns the binary representation of it in a character variable

- After it was loaded by root, all required files with their content broadcasted to fill their respective tables of the special require function

- Finally broadcast the main script and execute it on all processes (will execute requires but get the code from memory instead of from the file system)

soi_config_module

- With this approach the Lua script is executed by all processes and the same configuration state becomes available for all of them

- Encapsulated in the soi_config_module with the soi_config_open routine

Interacting with the User in MPI-Parallel Applications

- Getting configuration parameters from the user
- **Informing the user about progress and what is being done (logging)**
- Reacting to errors that the application detects, and reporting them to the user

Writing Log Information

- User needs feedback
  - Whether configuration settings are all correctly received by application
  - How far the simulation progressed
- Helping developers to identify problems
- Feedback might be required with different levels of detail
- Usually not required by all processes

Log Information in Parallel

- Need to filter out log messages from most processes
- Still provide possibility to obtain loggings from multiple processes for debugging
- Separation of logs from multiple processes

Soi_log_module

- Different levels of detail:
  - Provide an array of file unit numbers to write to
  - Level of detail equivalent to index in the array of file units
  - Higher levels indicate less importance of the message

- Application verbosity can be limited at compile-time by setting a maximal logging level to consider

- Log command provided by a Fypp macro

Soi_log_module Configuration

- Level of detail is configurable at runtime
  - (up to maximal level of the compiled executable)

- Number of processes that will write a log can be configured
  - Only root (MPI_COMM_WORLD rank 0) will write to stdout, but may be configured to write to a file instead
  - Other processes only will write a log to a file if accordingly configured

- Formatting: Line length of log messages will be limited, limit can be set by user at runtime

Discarding Undesired Log Messages

- The file-unit array of the logger is filled with a unit connected to an appropriate file (stdout or configured filename) up to the level configured by the user all higher units are connected to the null device (/dev/null)
  - Example level=3: `funit=[stdout, stdout, stdout, null, null, null, …]`
  - funit filled at runtime after reading user settings
  - On processes that are not to write logs, all entries point to null

- The log macro will write to the unit found in the funit array
  - Example: **log(4, message)** -> `write(funit(4),*) message`

Example Logging Code

```
@:log(1, 'This application does not really do anything.')
@:log(1, 'But it shows how the basic configuration is loaded')
@:log(1, 'by soi_world_init.')
$:log_blank(1)
@:warn('Warnings will be colored!')
@:warn('They are always written on log level 1.')
$:log_blank(1)
$:log_sep(1)
$:log_indent()
@:log(1, 'Math constants:')
$:log(1, "'e  = ', exp(1.0)", log_fmt="'(a,f16.10)'")
$:log(1, "'Pi = ', acos(-1.0)", log_fmt="'(a,en16.9)'")
$:log_unindent()
@:log(2, 'A less important message, put on logging level 2.')
```

Example Logging Output

```
This application does not really do anything.
But it shows how the basic configuration is loaded
by soi_world_init.

Warnings will be colored!
They are always written on log level 1.

**************************************************************************
  Math constants:
  e  =     2.7182817459
  Pi =  3.141592741E+00
**************************************************************************
A less important message, put on logging level 2.
```

Remarks on the Approach

- All writes are local to the processes, no communication

- Two stages:
  - Compile time limitation of maximal log level allows minimization of running time impacts
  - Runtime configuration enables the user to set the desired level of verbosity

- Access to `/dev/null` by most processes should be fast and not limit scalability

Interacting with the User in MPI-Parallel Applications

- Getting configuration parameters from the user
- Informing the user about progress and what is being done (logging)
- **Reacting to errors that the application detects, and reporting them to the user**

Dealing with Errors

- Many thanks to Rolf Rabenseifner for suggesting the MPI strategy

- During parallel execution any process might run into an erroneous state
  - But not all processes may run into it

- We still want to properly end the simulation
  - Provide the user with a proper error notification
  - Possibly provide some data dump for inspection

Problem with Errors in Parallel Runs

- Any process may run into an error
- All processes need to be notified of this to coordinate program termination
- How to deal with this in MPI parallel applications?
  - Mainly two options:
    - One-sided communication
    - Non-blocking collectives

Handling Errors with Non-Blocking Collectives

- soi_error_module

- Basic idea:
  - Conditional raising of error (similar to log macro), in case of error
  - Subsequent unconditional checkup on error notification

- To achieve this:
  - Need an MPI_Ibcast on a dedicated MPI communicator (duplicate of MPI_COMM_WORLD)

Soi_error_module

- Startup:
  - Process 0 opens MPI_Irecv for MPI_ANY_SOURCE
  - All other processes start MPI_Ibcast with rank 0 as root

- In case of error:
  - Process with error sends message to rank 0

- Regularly checkup on possibly occured errors (unconditionally after conditional error raising)
- Finalization in case of error

Check on Errors

- Checking for occurred errors involves:
  - On process 0
    - checking the MPI_Irecv for completion
    - if message received, post the MPI_Ibcast to complete it, then wait on it and afterward start the abort analysis and processing
  - On other processes
    - Check the MPI_Ibcast for completion
    - If MPI_Ibcast completed, enter abort analysis

Example for Error Handling in Code

```
call aot_get_val( L       = lua,          &
  &               thandle = thandle,      &
  &               val     = config%level, &
  &               key     = 'level',      &
  &               default = level,        &
  &               ErrCode = iError        )
if (btest(iError, aoterr_Fatal)) then
  ! Conditional error throwing
  @:error('Error reading level for logging!')
  @:error('Level needs to be an integer, please fix your config.')
end if
…
! Unconditional checking for error
call soi_error_check()
```

Abort Analysis

- If there was an error:
  - Gather error messages from all processes **if sufficient memory** on root process
  - Report messages with the originating MPI rank:
    - Collapse contiguous ranks with the same message
  - If memory on root process insufficient for messages from all ranks, just get and print the longest error message to report
  - Finalize MPI and stop application

Example Output

- Same error on all 4 processes:

```
An error occurred!
Error messages on processes 0-3:
 Error reading level for logging!
 Level needs to be an integer, please fix your config.
```

Summary

- Dealing with minor IO tasks
  - Still important to be treated properly for scalable applications
- Dealing with errors in parallel applications to provide concise and reasonable messages if possible
- Convenience
  - For developers
  - For users

Thank You for Your Kind Attention!

Thanks to Rolf Rabenseifner for his support.

May I take questions?