Table of Contents

Parallel I/O with SIONlib

Modules

Recent versions of SIONlib are available on the DEEP-EST prototype through the new software stack as modules:

```
$ module spider SIONlib/1.7.6
```

SIONlib documentation

Documentation for SIONlib can be found on the web at ?https://apps.fz-juelich.de/jsc/sionlib/docu/current/index.html

MSA aware collective I/O

Recent versions of SIONlib contain mechanisms to perform I/O operations collectively, i.e. all processes of a parallel computation partake in these operations. This enables an exchange of I/O data between the processes, allowing a subset of all processes, the *collector* processes, to perform the actual transfer of data to the storage on behalf of other processes. Collector processes should typically be those processes that are placed on parts of the MSA with a high bandwidth connection to the parallel file system. The mechanism has been extended with a new MSA-aware algorithm for the selection of collector processes. The algorithm is portable and relies on platform specific plug-ins to identify processes which run on parts of the system that are well suited for the role of I/O collector. So far, three plug-ins have been implemented:

- a new generic plug-in — `hostname-regex` — which selects collectors tasks by matching the host name of the node they are running on against regular expressions that can be defined via environment variables,
- an older plug-in — `deep-est-std` — that selects collector tasks using a similar mechanism, but with host names hard-coded to match cluster nodes of an earlier version of the DEEP-EST prototype,
- a mock-up plug-in for testing purposes.

In the future, further plug-ins for new systems of the MSA type can be added.

In order to use the MSA aware collective I/O operations, a platform specific plug-in has to be selected when installing SIONlib during the configure step.

```
./configure --msa=hostname-regex # ... more configure arguments
```

When opening a SIONlib file for access from several MPI processes in parallel, the user has to enable the MSA aware collective I/O mode. This is done using the `file_mode` argument of the open function. `file_mode` contains a string that consists of a comma separated list of keys and key value pairs. The word `collmsa` must appear in that list to select MSA aware collective I/O.

```
sion_paropen_mpi("filename", "...,collmsa,...", ...);
```

Also, when the new `hostname-regex` collector selection plug-in is enabled, an environment variable `SION_MSA_COLLECTOR_HOSTNAME_REGEX` has to be defined to contain a POSIX basic regular expression to match against hostnames of nodes running candidates for the collector role (`SION_MSA_COLLECTOR_HOSTNAME_EREGEX` can be used alternatively, but should contain a POSIX extended regular expression). For example, to select nodes from the DAM as collectors, set:

```
export SION_MSA_COLLECTOR_HOSTNAME_EREGEX="dp-dam.*"
```

Additionally, as is the case when using regular collective I/O, the size of collector groups has to be secified, either through the `file_mode` argument of the `sion_paropen_mpi` function or via the environment variable `SION_COLLSIZE`.

SIONlib CUDA aware interface

In order to match the programming interface offered by other libraries, such as ParaStation? MPI, more closely, functions of SIONlib have been made CUDA-aware. This means that applications are allowed to pass device pointers pointing to on device memory to the various read and write functions of SIONlib without needing to manually copy their contents to the host memory.

Like the MSA aware collective I/O operations, the CUDA aware interface has to be enabled when installing SIONlib. This is done by invoking the `configure` script with the argument `--enable-cuda` which optionally allows to specify the path to a CUDA installation.

```
./configure --enable-cuda=/path/to/cuda/installation # ... more configure arguments
```

When SIONlib has been installed with the CUDA aware interface enabled, the user may pass device pointers as the `data` argument to SIONlib's

- task-local
- key/value
- collective

read and write functions.

```
size_t sion_fwrite(const void *data, size_t size, size_t nitems, int sid);
size_t sion_fread(void *data, size_t size, size_t nitems, int sid);
size_t sion_fwrite_key(const void *data, uint64_t key, size_t size, size_t nitems, int sid);
size_t sion_fread_key(void *data, uint64_t key, size_t size, size_t nitems, int sid);
size_t sion_coll_fwrite(const void *data, size_t size, size_t nitems, int sid);
size_t sion_coll_fread(void *data, size_t size, size_t nitems, int sid);
```

SIONlib inspects the pointer and if it points to an on-device buffer performs a block-wise copy of the data into host memory before writing to disk or into device memory after reading from disk.

I/O forwarding

MSA aware collective I/O has the potential of making more efficient use of the storage system by using a subset of tasks that are well suited for performing I/O operations as collectors. The collective I/O approach however imposes additional constraints that make it inapplicable in certain scenarios:

- By design, collective I/O operations force application tasks to coordinate in order to all perform the same sequence of operations. This is at odds with SIONlib's world view of separate files per task that can be accessed independently.
- Collector tasks in general have to be application tasks, i.e. they have to run the user's application. This can generate conflicts on MSA systems, if the nodes that are capable of performing I/O operations efficiently are part of a module that the user application does not map well onto.

I/O forwarding can help in both scenarios. It works by relaying calls to low-level I/O functions (e.g. `open`, `write`, `stat`, etc.) via a remote procedure call (RPC) mechanism from a client task (running the user's application) to a server task (running a dedicated server program) that then executes the functions on behalf of the client. Because the server tasks are dedicated to performing I/O, they can dynamically respond to individual requests from client tasks rather than imposing coordination constraints. Also, on MSA systems, the server tasks can run on different modules than the user application.

I/O forwarding has been implemented in SIONlib through an additional software package, SIONfwd (?https://gitlab.version.fz-juelich.de/SIONlib/SIONfwd). It consists of a server program and a corresponding client library that is used by SIONlib to relay the low-level I/O operations that it wants to perform to the server. The implementation uses a custom made, minimal RPC mechanism based only on MPI's message passing, ports, and pack/unpack mechanisms. In the future we intend to evaluate more general and more optimised third party RPC solutions as they become available.

To use I/O forwarding in SIONlib, the SIONfwd package first has to be installed (it uses a standard CMake based build system) and SIONlib has to be configured to make use of it:

{{{!#sh ./configure —enable-sionfwd=/path/to/sionfwd # … more configure arguments }}}

In the user application, just like MSA aware collectives, I/O forwarding has to be selected when opening a file (I/O forwarding is treated like an additional low-level API like POSIX and C standard I/O). This is done by adding the word `sionfwd` to the `file_mode` argument of SIONlib's `open` functions:

```
sion_paropen_mpi("filename", "...,sionfwd,...", ...);
```

Although in principle MPI contains a mechanism for dynamically spawning additional processes, it is not used to spawn the forwarding server processes for two reasons. First, the feature is loosely specified with many of the details left for implementations to decide. This makes it hard to precisely control process placement which is especially important on MSA systems. Second, the resources necessary to run the server tasks (additional compute nodes) in many cases have to be requested at job submission time anyway. Thus, the server tasks have to be launched from the user's job script before the application tasks are launched. A typical job script could look like this:

```
#!/bin/bash
# Slurm's heterogeneous jobs can be used to partition resources
# for the user's application and the forwarding server, even
# when not running on an MSA system.
#SBATCH --nodes=32 --partition=dp-cn
```

```
#SBATCH packjob
#SBATCH --nodes=4 --cpus-per-taks=1 --partition=dp-dam

module load intel-para SIONlib/1.7.6

# Defines a shell function sionfwd-spawn that is used to
# facilitate communication of MPI ports connection details
# between the server and the client.
eval $(sionfwd-server bash-defs)

# Spawns the server, captures the connection details and
# exports them to the environment to be picked up by the
# client library used from the user's application.
sionfwd-spawn srun --pack-group 1 sionfwd-server

# Spawn the user application.
srun --pack-group 0 user_application

# Shut down the server.
srun --pack-group 0 sionfwd-server shutdown

# Wait for all tasks to end.
wait
```