Developing XIOS with multi-thread : to accelerate the I/O of climate models

July 13, 2018

1 Context

The simulation models of climate systems, running on a large number of computing resources can produce an important volume of data. At this scale, the I/O and the post-treatment of data becomes a bottle-neck for the performance. In order to manage efficiently the data flux generated by the simulations, we use XIOS developed by the Institut Pierre Simon Laplace and Maison de la simulation.

XIOS, a library dedicated to intense calculates, allows us to easily and efficiently manage the parallel I/O on the storage systems. XIOS uses the client/server scheme in which computing resources (server) are reserved exclusively for IO in order to minimize their impact on the performance of the climate models (client). The clients and servers are executed in parallel and communicate asynchronously. In this way, the I/O peaks can be smoothed out as data fluxes are send to server constantly throughout the simulation and the time spent on data writing on the server side can be overlapped completely by calculates on the client side.

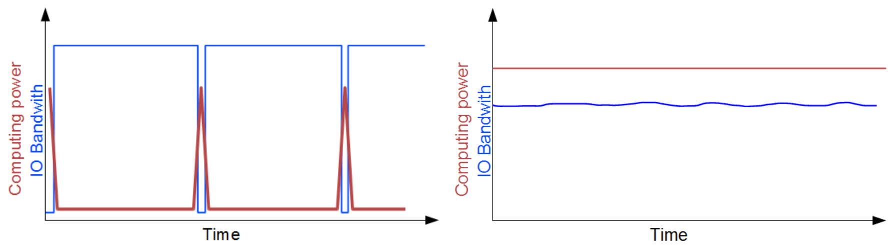

Figure 1: On the left, each peak of computing power corresponds to the valley of memory bandwidth which shows that the computing resources are alternating between calculates and I/O. ON the right, both curves are smooth which means that the computing resources have a stable charge of work, either calculates or I/O.

XIOS works well with many climate simulation codes. For example, LMDZ[1],

[1]LMDZ is a general circulation model (or global climate model) developed since the 70s at the "Laboratoire de Météorologie Dynamique", which includes various variants for the Earth and other planets (Mars, Titan, Venus, Exoplanets). The 'Z' in LMDZ stands for "zoom" (and the 'LMD' is for 'Laboratoire de Météorologie Dynamique"). http://lmdz.lmd.jussieu.fr

NENO[2], ORCHIDEE[3], and DYNAMICO[4] all use XIOS as the output back end. MétéoFrance and MetOffice also choose XIOS to manage the I/O for their models.

2 Development of thread-friendly XIOS

Although XIOS copes well with many models, there is one potential optimization in XIOS which needs to be investigated: making XIOS thread-friendly.

This topic comes along with the configuration of the climate models. Take LMDZ as example, it is designed with the 2-level parallelization scheme. To be more specific, LMDZ uses the domain decomposition method in which each sub-domain is associated with one MPI process. Inside of the sub-domain, the model also uses OpenMP derivatives to accelerate the computation. We can imagine that the sub-domain be divided into sub-sub-domain and is managed by threads.

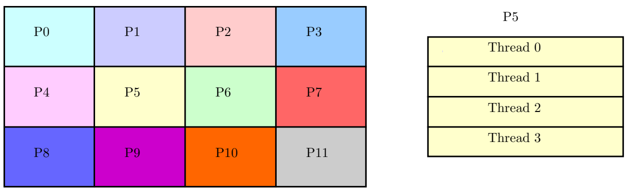

Figure 2: Illustration of the domain decomposition used in LMDZ.

As we know, each sub-domain, or in another word, each MPI process is a XIOS client. The data exchange between client and XIOS servers is handled by MPI communications. In order to write an output field, all threads must gather the data to the master thread who acts as MPI process in order to call MPI routines. There are two disadvantages about this method : first, we have to spend time on gathering information to the master thread which not only increases the memory use, but also implies an OpenMP barrier; second, while the master thread calls MPI routine, other threads are in the idle state thus a waster of computing resources. What we want obtain with the thread-friendly XIOS is that all threads can act like MPI processes. They can call directly the MPI routine thus no waste in memory nor in computing resources as shown in Figure 3.

There are two ways to make XIOS thread-friendly. First of all, change the structure of XIOS which demands a lot of modification is the XIOS library. Knowing that XIOS is about 100 000 lines of code, this method will be very time consuming. What's more, the modification will be local to XIOS. If we want to optimize an other code to be thread-friendly, we have to redo the modifications. The second choice is to add an extra interface to MPI in order to manage the

[2]Nucleus for European Modeling of the Ocean alias NEMO is a state-of-the-art modelling framework of ocean related engines. `https://www.nemo-ocean.eu`

[3]the land surface model of the IPSL (Institut Pierre Simon Laplace) Earth System Model. `https://orchidee.ipsl.fr`

[4]The DYNAMICO project develops a new dynamical core for LMD-Z, the atmospheric general circulation model (GCM) part of IPSL-CM Earth System Model. `http://www.lmd.polytechnique.fr/~dubos/DYNAMICO/`

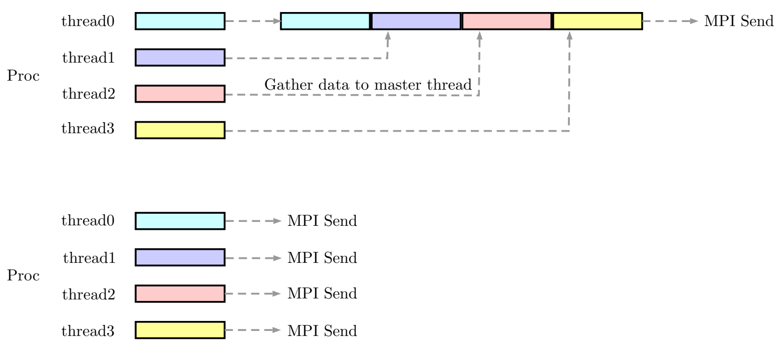

Figure 3:

threads. When a thread want to call an MPI routine inside XIOS, it will first pass the interface, in which the communication information will be analyzed before the MPI routine is invoked. With this method, we only need to modify a very small part of XIOS in order to make it work. What is more interesting is that the interface we created can be adjusted to suit other MPI based libraries.

In this project, we choose to implement an interface to handle the threads. To do so, we introduce the MPI_endpoint which is a concept proposed in the last MPI Forums and several papers have already discussed the importance of such idea and have introduced the framework of the MPI_endpoint [2][3]. The concept of an endpoint is shown by Figure 4. In the MPI_endpoint environment, each OpenMP thread will be associated with a unique rank (global endpoint rank), an endpoint communicator, and a local rank (rank inside the MPI process) which is very similar to the `OMP_thread_num`. The global endpoint rank will replace the role of the classic MPI rank and will be used in MPI communication calls.

An other important aspect about the MPI_endpoint interface is that each endpoints has knowledge of the ranks of other endpoints in the same communicator. This knowledge is necessary because when executing an MPI communication, for example a point-to-point exchange, we need to know not only the ranks of sender/receiver threads, but also the thread number of the sender/receiver threads and the MPI ranks of the sender/receiver processes. This ranking information is implemented inside an map object included in the endpoint communicator class.

In XIOS, we used the "probe" technique to search for arrived messages and then perform the receiving action. The principle is that sender process executes the send operation as usual. However, to minimize the time spent on waiting incoming messages, the receiver process calls in the first place the `MPI_Probe` function to check if a message destinate to it has been published. If yes, the process execute in the second place the `MPI_Recv` to receive the message. If not, the receiver process can carry on with other tasks or repeats the `MPI_Probe` and `MPI_Recv` actions if the required message is in immediate need. This technique works well in the current version of XIOS. However, if we introduce threads into this mechanism, problems can occur: The incoming message is labeled by the tag and receiver's MPI rank. Because threads within a process share the MPI rank, and the message probed is always available in the message queue, it can

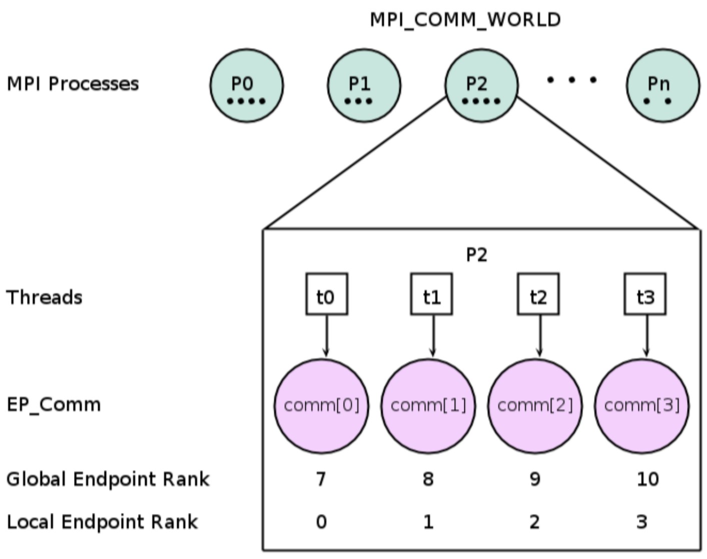

Figure 4:

lead to the problem of data race and thus the message can be received by the wrong thread.

To solve this problem, we introduce the “matching-probe” technique. The idea of the method is that each thread is equipped with a local incoming message queue. Each time a thread calls an MPI function, for example `MPI_Recv`, it calls firstly the `MPI_Mprobe` function to query the MPI incoming message with any tag and from any source. Once a message is probed, the thread gets the handle to the incoming message and this specific message is erased from the MPI message queue. Then, the thread proceed the identification of the message to get the destination thread’s local rank and store the message handle to the local queue of the target thread. The thread repeats these steps until the MPI incoming message queue is empty. Then the thread we perform the usual “probe” technique to query its local incoming message queue to check if the required message is available. If yes, it performs the `MPI_Recv` operation. With this “matching-probe” technique, we can assure that a message is probed only once and is received by the right receiver.

Another issue needs to be clarified with this technique is that: how to identify the receiver’s rank? The solution to this question is to use the tag argument. In the MPI environment, a tag is an integer ranging from 0 to 2^{31} depending on the Implementation. We can explore the large range property of the tag to store in it information about the source and destination thread ranks. In our endpoint interface, we choose to limit the first 15 bits for the tag used in the classic MPI communication, the next 8 bits to store the sender thread’s local rank, and the last 8 bits to store the receiver thread’s local rank (*c.f.* Figure 5). In this way, with an extra analysis of the tag, we can identify the local ranks of the sender and the receiver in any P2P communication. As results, when a thread probes a message, it knows exactly in which local queue should store the probed message.

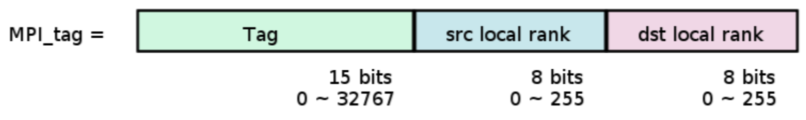

Figure 5:

In Figure 5, Tag contains the user defined value for a certain communication. MPI_tag is computed in the endpoint interface with help of the rank map and is used in the MPI calls.

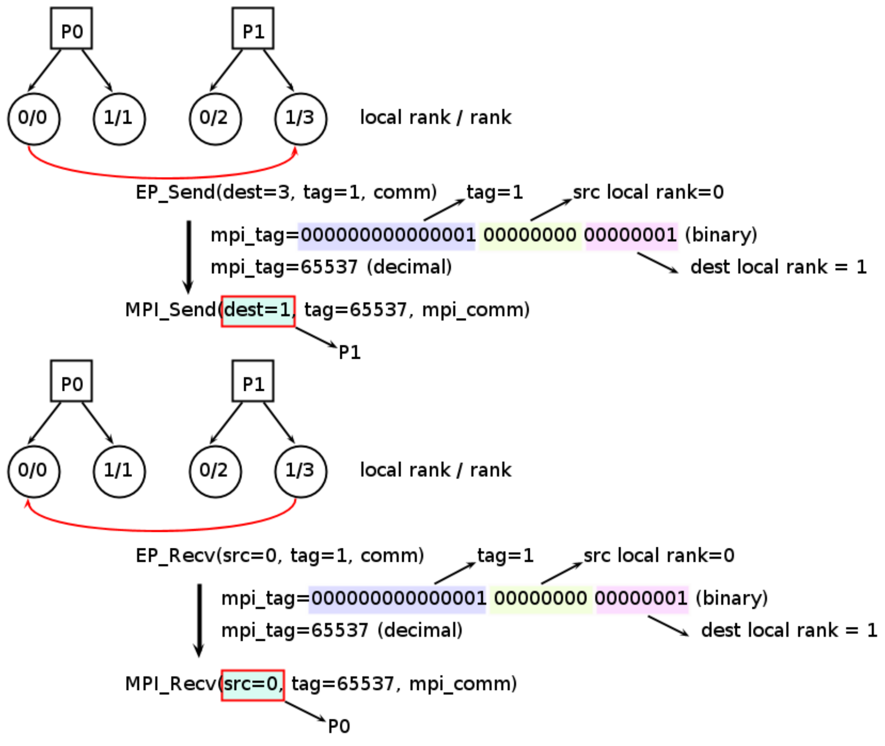
P0
P1
0/0
1/1
0/2
1/3
local rank / rank
EP_Send(dest=3, tag=1, comm)
tag=1
src local rank=0
mpi_tag=000000000000001 00000000 00000001 (binary)
mpi_tag=65537 (decimal)
dest local rank = 1
MPI_Send(dest=1, tag=65537, mpi_comm)
P1
P0
P1
0/0
1/1
0/2
1/3
local rank / rank
EP_Recv(src=0, tag=1, comm)
tag=1
src local rank=0
mpi_tag=000000000000001 00000000 00000001 (binary)
mpi_tag=65537 (decimal)
dest local rank = 1
MPI_Recv(src=0, tag=65537, mpi_comm)
P0

Figure 6: This figure shows the classic pattern of a P2P communication with the endpoint interface. Thread/endpoint rank 0 sends a message to thread/endpoint rank 3 with tag=1. The underlying MPI function called by the sender is indeed a send for MPI rank of 1 and tag=65537. From the receiver's point of view, the endpoint 3 is actually receiving a message from MPI rank 0 with tag=65537.

With the rank map, tag extension, and the matching-probe techniques, we are now able to call any P2P communication in the endpoint environment. For the collective communications, we apply a step-by-step execution pattern and no special technique is required. A step-by-step execution pattern consists of 3 steps (not necessarily in this order and not all steps are needed): arrangement of the source data, execution of the MPI function by all master/root threads, distribution or arrangement of the resulting data among threads.

For example, if we want to perform a broadcast operation, only 2 steps are needed (*c.f.* Figure 7). Firstly, the root thread, along with the master threads of other processes, perform the classic `MPI_Bcast` operation. Secondly, the root thread, and the master threads send data to other threads via local memory transfer.

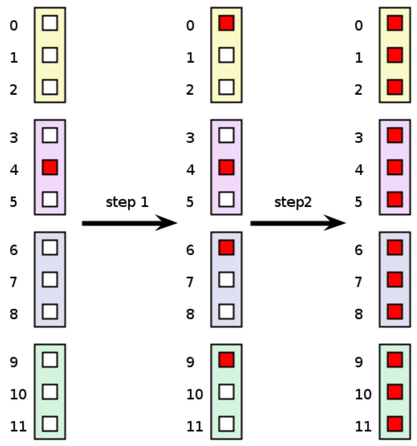

Figure 7: `MPI_Bcast`

Figure 8 illustrates how the `MPI_Allreduce` function is proceeded in the endpoint interface. First of all, We perform a intra-process "allreduce" operation: source data is reduced from slave threads to the master thread via local memory transfer. Next, all master threads call the classic `MPI_Allreduce` routine. Finally, all master threads send the updated reduced data to its slaves via local memory transfer.

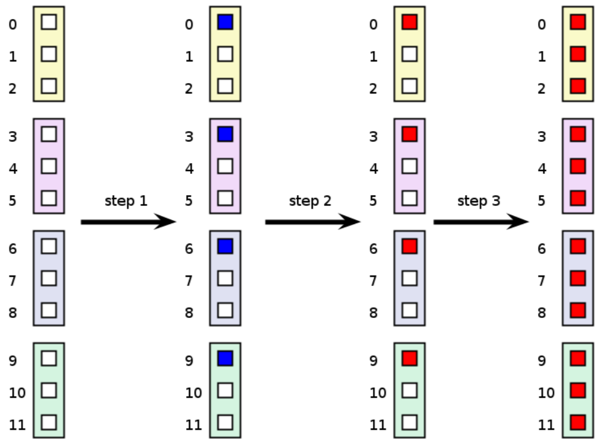

Figure 8: `MPI_Allreduce`

Other MPI routines, such as `MPI_Wait`, `MPI_Intercomm_create` *etc.*, can be found in the technique report of the endpoint interface [1].

3 The multi-threaded XIOS and performance results

The development of endpoint interface for thread-friendly XIOS library took about one year and a half. The main difficulty is the co-existence of MPI processes and OpenMP threads. One essential requirement for using the endpoint interface is that the underlying MPI implementation must support the level-3 of thread support which is `MPI_THREAD_MULTIPLE`. This means that if the MPI process is multi-threaded, multiple threads may call MPI at once with no restrictions. Another importance aspect to be mentioned is that in XIOS, we have variables with `static` attribute. It means that inside of an MPI process, threads share the static variable. In order to use correctly the endpoint interface, these static variables have to be defined as `threadprivate` to limit the visibility to thread.

To develop the endpoint interface, we redefined all MPI classes along with all the MPI routines that are used in XIOS library. The current version of the interface includes about 7000 lines of code and is now available on the forge server: `http://forge.ipsl.jussieu.fr/ioserver/browser/XIOS/dev/branch_openmp`. One technique report is also available in which one can find more detail about how endpoint works and how the routines are implemented [1]. We must note that the thread-friendly XIOS library is still in the phase of optimization. It will be released in the future with a stable version.

All the functionalities of XIOS is reserved in its thread-friendly XIOS library. Single threaded code can work successfully under the endpoint interface with the new version of XIOS. For multi-threaded models, some modifications are needed in order to work with the multi-threaded XIOS library. For example, the MPI initialization has be to modified to require the `MPI_THREAD_MULTIPLE` support. Each thread should have its own data set. What's most important is that the OpenMP master region in which the master thread calls XIOS routines should be erased in order that every threads can call XIOS routines simultaneously. More detail can be found in our technique report [1].

Even though the multi-threaded XIOS library is not fully accomplished and further optimization in ongoing. We have already done some tests to see the potential of the endpoint framework. We take LMDZ as the target model and have tested with several work-flow charges.

3.1 LMDZ work-flow

In the LMDZ work-flow, we have a daily output file. We have up to 413 two-dimension variables and 187 three-dimension variables. According to user's need, we can change the "output_level" key argument in the `xml` file to select the desired variables to be written. In our tests, we choose to set "output_level=2" for a light output, and "output_level=11" for a full output. We run the LMDZ code for one, two, and three-month simulations using 12 MPI client processes and 1 server process. Each client process includes 8 OpenMP threads which gives us 92 XIOS clients in total.

3.2 CMIP6 work-flow

4 Future works for XIOS

References

[1] XIOS developper group. Note for XIOS Endpoints. Technical report, `http://forge.ipsl.jussieu.fr/ioserver/browser/XIOS/dev/branch_openmp/Note`, 2018.

[2] J. Dinan, Pavan Balaji, D. Goodell, D. Miller, M. Snir, and Rajeev Thakur. Enabling MPI Interoperability Through Flexible Communication Endpoints. In *EuroMPI 2013*, Madrid, Spain, 2013.

[3] S. Sridharan, J. Dinan, and D. D. Kalamkar. Enabling Efficient Multithreaded MPI Communication Through a Library-Based Implementation of MPI Endpoints. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 487–498, Nov 2014.