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

June 25, 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.

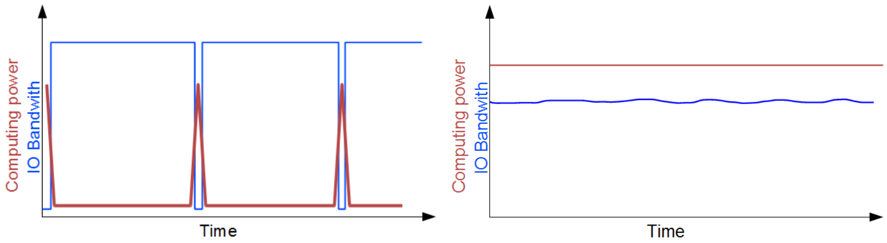

Figure 1: On the left, each peak of computing power corresponds to the valley of memory bandwidth which shows that the computing resources are alternating between calculates and I/O. ON the right, both curves are smooth which means that the computing resources have a stable charge of work, either calculates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

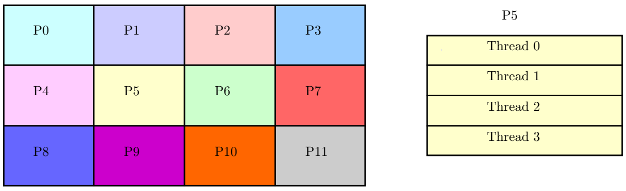

Figure 2: Illustration of the domain decomposition used in LMDZ.

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/`

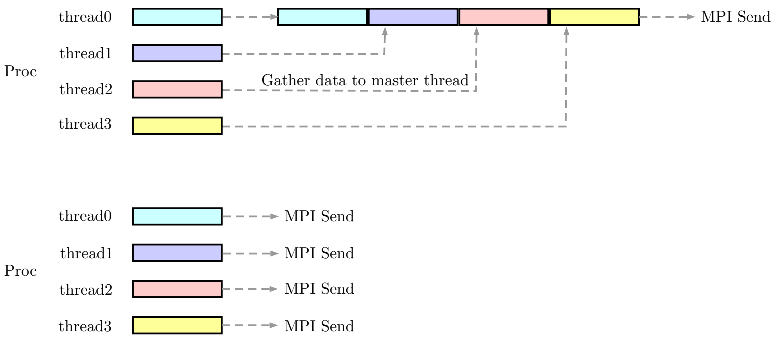

Figure 3:

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 the interface to handle the threads. To do so, we introduce the MPI_endpoint which is a concept proposed in the last MPI Forum and several papers has already discussed the importance of such idea and have introduced the framework of the MPI_endpoint [1][2]. The concept of an endpoint is shown by Figure 4. Threads of an MPI process is associated with a unique rank (global endpoint rank) and an endpoint communicator. They also have a local rank (rank inside the MPI process) which is very similar to the `OMP_thread_num` rank.

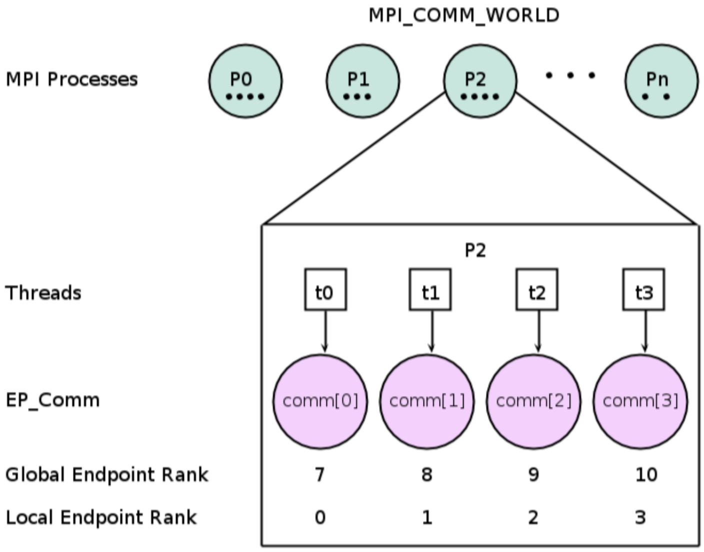

Figure 4:

The MPI endpoints (EP) is a layer on top of an existing MPI Implementation. All MPI function, or in our work the functions used in XIOS, will be re-implemented in order to cope with OpenMP threads. The idea is that, in the MPI endpoints environment, each OpenMP thread will be associated with a unique rank and with an endpoint communicator. This rank (EP rank) will replace the role of the classic MPI rank and will be used in MPI communications. In order to successfully execute an MPI communication, for example `MPI_Send`, we know already which endpoints to be the receiver but not sufficient. We also need to know which MPI process should be involved in such communication. To identify the MPI rank, we added a “map” in the EP communicator in which the relation of all EP and MPI ranks can be easily obtained.

In XIOS, we used the “probe” technique to search for arrived messages and then performing the receive action. The principle is that sender processes execute the send operations as usual. However, to minimise the time spent on waiting incoming messages, the receiver processe performs in the first place the `MPI_Probe` function to check if a message destinated to it has been published. If yes, the process execute in the second place the `MPI_Recv` to receive the message. In this situation, if we introduce the threads, problems occur. The reason why the “probe” method is not suitable is that messages destinated to one certain process can be probed by any of its threads. Thus the message can be received by the wrong thread which gives errors.

To solve this problem, we introduce the “matching-probe” technique. The idea of the method is that each process is equiped with a local incoming message queue. All incoming message will be probed, sorted, and then stored in this queue according to their destination rank. Every time we call an MPI function, we firstly call the `MPI_Mprobe` function to get the handle to the incoming message. Then, we identify the destination thread rank and store the message handle inside the local queue of the target thread. After this, we perform the usual “probe” technique upon the local incoming message queue. In this way, we can assure the messages to be received by the right thread.

Another issue remains in this technique: how to identify the receiver's rank? The solution is to use the tag argument. In the MPI environment, a tag is an integer ranging from 0 to 2^{31}. We can explore the large range of the tag to store in it information about the source and destination thread ranks. We choose to limite the first 15 bits for the tag used in the classic MPI communication, the next 8 bits to the sender's thread rank, and the last 8 bits to the receiver's thread rank. In such way, with an extra analysis of the EP tag, we can identify the ranks of the sender and the receiver in any P2P communication. As results, we a thread probes a message, it knows exactly in which local queue should store the probed message.

With the global rank map, tag extension, and the matching-probe techniques, we are able to use any P2P communication in the endpoint environment. For the collective communications, we perform a step-by-step execution and no special technique is required. The most representative functions is the collective communications are `MPI_Gather` and `MPI_Bcast`. A step-by-step execution consists of 3 steps (not necessarily in this order): arrangement of the source data, execution of the MPI function by all master/root threads, distribution or arrangement of the data among threads.

For example, if we want to perform a broadcast operation, 2 steps are needed. 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 threads sharing the same process via local memory transfer. In another example for illustrating the `MPI_Gather` function, we also need 2 steps. First of all, data is gathered from slave threads to the master thread or the root thread. Next, the master thread and the root thread execute the `MPI_Gather` operation of complete the communication. Other collective calls such as `MPI_Scan`, `MPI_Reduce`, `MPI_Scatter` *etc* follow the same principle of step-by-step execution.

3 Performance of LMDZ using EP_XIOS

With the new version of XIOS, we are now capable of taking full advantages of the computing resources allocated by a simulation model when calling XIOS functions. All threads, can participate in XIOS as if they are MPI processes. We have tested the EP_XIOS in LMDZ and the performance results are very encouraging.

In our tests, we used 12 client processor with 8 threads each (96 XIOS clients in total), and one single-thread server processor. We have 2 output densities. The light output gives mainly 2 dimensional fields while the heavy output records more 3D fields. We also have differente simulation duration settings: 1 day, 5 days, 15 days, and 31 days.

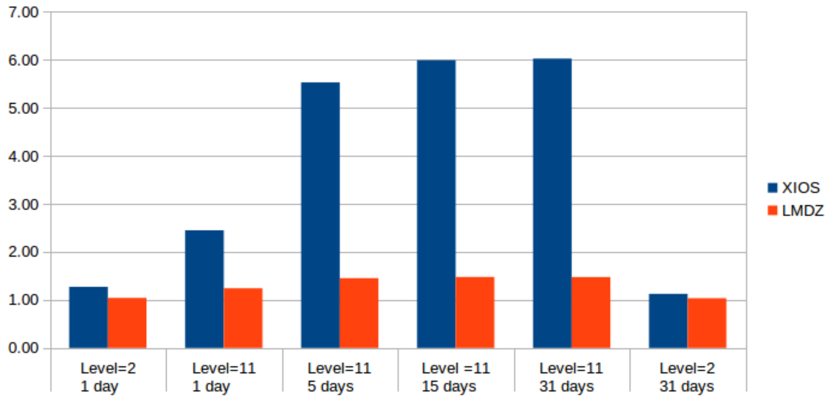

Figure 5: Speedup obtained by using EP in LMDZ simulations.

In this figure, we show the speedup which is computed by $\frac{time_{XIOS}}{time_{EP\_XIOS}}$. The blue bars represent speedup of the XIOS file output and the red bars the speedup of LMDZ: calculates + XIOS file output. In all experimens, we can observe a speedup which represents a gain in performance. One important conclusion we can get from this result is that, more dense the output is, more efficient is the EP_XIOS. With 8 threads per process, we can reach a speedup in XIOS upto 6, and a speedup of 1.5 in LMDZ which represents a decrease of the total execution time to 68% (≈ 1/1.5). This observation confirmes steadily the importance of using EP in XIOS.

The reason why LMDZ does not show much speedup, is because the model is calcutation dominant: time spent on calculation is much longer than that on the file output. For example, if 30% of the execution time is spent on the output, then with a speepup of 6, we can obtain a decrease in time of 25%. Even the 25% may seems to be small, it is still a gain in performance with existing computing resources.

4 Performance of EP_XIOS

workfloz_cmip6 light output 24*8+2 30s - 52s 32 days histmth with daily output

5 Perspectives of EP_XIOS

References

[1] 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.

[2] S. Sridharan, J. Dinan, and D. D. Kalamkar. Enabling efficient multi-threaded 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.