Grasset, Judicaël; Audouin, Yoann; Fontaine, Jacques; Moulinec, Charles; Emerson, David R.

Improving TELEMAC system pre-processing and IO stages

Improving TELEMAC system pre-processing and IO stages

Judicaël Grasset[1,2],Yoann Audouin[1], Jacques Fontaine[1], Charles Moulinec[2], David R. Emerson[2]
[1]EDF R&D, LNHE, Chatou, 78400, France
[2]STFC Daresbury Laboratory, Sci-Tech Daresbury, Warrington, WA4 4AD, United Kingdom

Abstract— **Improvements in the pre-processing (partel serial) and IO stages are presented to facilitate running the suite of software on high-end machines. Firstly we present how the memory consumption of partel serial has been decreased and secondly, how partel serial can now generate only a set of files for all the MPI processes instead of one for each MPI process. Finally a work in progress dealing with the reduction of result files generated by a run of TELEMAC is presented.**

I. INTRODUCTION

In order to prepare TELEMAC-MASCARET for very large simulations, we present here some improvements in the serial version of partel to facilitate running the suite of software on high-end machines:

- We have drastically reduced the peak memory consumption of the serial pre-processor partel, as the current version of partel uses a huge amount of memory to pre-process meshes, which makes it impossible to use it for large meshes (several 10s or 100s million elements) even with access to fat memory nodes. However, there exists a parallel version of partel in the TELEMAC distribution, to circumvent this memory issue. It is split into a first serial stage where METIS is used as a default partitioner and where the mesh nodes/elements are allocated to their respective subdomain; and a second stage, which is parallel, where the rest of the pre-processing is carried out. Note that for very large meshes, the first stage still requires a lot of memory, whereas the second stage requires the same number of processors as the number of subdomains to be used, which makes the whole process tedious.

- We have reduced the number of files generated by partel serial; the current version of partel generates a set of files for each MPI process. This is an issue when using a large number of MPI processes as the high-end machine operating systems have a hard limit on the number of files a single user can open.

- We have worked to reduce the number of files generated by the solvers themselves. The issue is very similar to the one highlighted in the second item, where the input files are generated by partel, although the solution is different. This is still work in progress.

How these improvements have been made is explained in the following sections.

II. THE HIGH END MACHINE

All the benchmarks presented here have been carried out using the UK National Facility ARCHER [3].

The ARCHER supercomputer is made of 4,544 compute nodes. Each of these compute nodes has two twelve cores Intel Ivy Bridge E5-2697v2 and 64 GiB of memory. These are the nodes used to carry out all the benchmarks presented in this article.

ARCHER has also 376 high memory nodes with 128 GiB of memory, but these nodes have not been used for the present benchmarks.

III. IMPROVING PARTEL

A. Reduce peak memory consumption

partel is the pre-processing tool used by the TELEMAC-MASCARET hydrodynamic suite of software. It is used to pre-process a mesh in order to distribute the simulation load accross the MPI processes.

A major problem with partel is that it requires a lot of memory if a mesh has more than 6 million elements. In order to understand the root of the problem, we have used the heap profiler called massif [7] from the Valgrind profiling suite. This tool can provide the exact line of the program where a faulty allocation occurs.

massif shows that the memory intensive arrays are CUT_P, KNOGL, GELEGL. After looking at the contents of these arrays, it appears that they are mostly used to store zeros, e.g. in some cases the arrays have a non-zero density of less than 0.01%!

In order to reduce the peak memory consumption, we need to save only the non-zero values while still being able to know at which position the zero values should be. To achieve this, a better data structure than the existing one is needed. Using a hash table seems to be the right solution. It provides a quick insert and lookup (O(1) on average) as well as a low memory footprint, because it uses a default value of zero (if a lookup fails, it means that the index is not in the table and then the default value, e.g. zero, is returned). As hash tables do not exist in the Fortran standard library, a custom one had to be implemented.

The hash table (see Appendix) can *insert*, *modify*, *lookup* but cannot *delete* because the original code does not remove

any element from the arrays. This functionality is thus not implemented. The *modify* function is simply done by a call to the *insert* function. It looks for the value and modifies it. The hash table grows by doubling its size (default size is 2^{20} at the creation of the hash table).

The arrays are in two dimensions, which means that the key for the hash table is a pair of integers. These two integers are paired together into a single one by using the Elegant pairing algorithm of Matthew Szudzik [1]. Then this integer is hashed with a modified version of a hash function taken from the Google FarmHash library [2].

The internal structure of the hash table is made of an array of structures which contains a pair of integers and an indicator to know if the element is used or not. The collisions – a collision happens when two different keys are hashed to the same value, and so they should be placed in the same cell of the hash table – are solved via linear probing, which means that if the cell is already taken, we put the value in the next free one. If this mechanism becomes too slow for some reasons in the future, we could reduce the maximum number of collisions by using a better algorithm or combining it with another method, as for instance, the robin hood hashing method [4].

The effect of using the hash table instead of the original big arrays can be seen in Fig. 1. The benchmark uses the *geo_malpasset-small.slf* mesh from the TELEMAC2D examples and METIS [5] is used for partitioning. The original mesh is globally refined several times using STBTEL (see Table 1), the new meshes are then partitioned using both versions of partel serial. The objective is for each partition to contain about 10k elements, which is a good estimation of the smallest number of elements to be used per subdomain for TELEMAC2D to still show good scalability on fast high-end machines. The two compared versions are the last stable version at the time of this work (V7P3R1) and the trunk revision 11,882 with the last patch for the hash table [6].

As the original partel requires too much memory we do not have enough measurements from the stable version to carry out the full comparison. However, we have added an estimation of the minimal memory required, which is obtained by summing up the memory needed by the three biggest arrays (CUT_P, GELEGL, KNOLG), as p*(2n+e)*i ; where p is the number of partitions, n is the number of grid nodes, e is the number of elements and i is the size of an integer (4 bytes in the code). Using this estimation we can perform the comparison and see that the peak memory consumption has been drastically reduced.

TABLE 1. MESH CHARACTERISTICS AFTER REFINING WITH STBTEL

Case #	1	2	3	4	5	6
No. elements	104K	416K	1.66M	6.65M	26.6M	106M
No. points	53K	210K	836K	3.33M	13.3M	53M

Case 4, see Table 1, does not require more than the maximum memory available on the compute node (about 33GiB vs 64GiB), but the simulation still crashes because one of the arrays (GELEGL) exceeds the maximum memory per allocation prescribed by the cluster administrator. For Cases 5 and 6, even if we assume that there is no maximum memory per allocation imposed, the original partel would still crash because the compute node does not have hundreds of GiBs of memory available. With the hash table version, the memory consumption is much less intensive and it is possible to partition a one hundred million element mesh without any issue. Note that the main source of peak memory consumption is now due to the use of the METIS library.

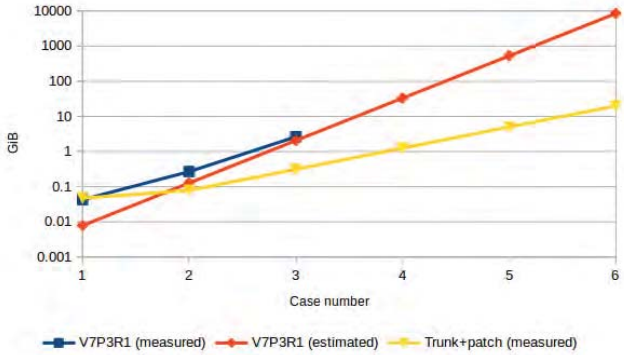

Figure 1. *Peak memory consumption of 2 different versions of partel for different size of meshes. (y-axis in logarithmic scale)*

We can also see on Fig. 1 that the estimation is below the measurements for the first two cases but is on par for the third case. This can be explained for the small cases by the memory consumption mostly coming from METIS and many small allocations. However, when the number of elements increases the memory consumption becomes dominated by the three aforementioned arrays.

During this optimisation stage, some useless iterations in some loops have been removed and the number of readings from the disk have also been reduced. These two modifications are completely independent from the hash table. However, Table 2 shows that they improve the execution time.

TABLE 2. COMPARISON OF THE EXECUTION TIME FOR BOTH VERSIONS OF PARTEL (IN SECONDS)

Case #.	1	2	3
V7P3R1	0.395	3.49	48.1
Trunk+patch	0.34	1.415	7.73

To replicate this experiment on your own cluster, you would need to download TELEMAC from the trunk and set it to the revision 11,882. You would also need to download the patch that contains the last version of the hash table and

the benchmarks script from [6]. Other details on how to apply the patch are available in the repository.

B. Reduce the number of files created

In the current version of the suite, each MPI process must have its own set of input files. These files are generated by partel from the original input files given by the user (mesh, boundary conditions and potential extra files to account for some more physics). For instance, the geometry file is divided into p files (one per MPI process). This division is repeated for every input file, which leads to at least X*p files where X is the number of original input files.

For example, for the TOMAWAC case called *opposing_current*, which requires 4 input files (WACGEO, WACCLI, WACPAR, WACCOB), if it is run using 250 MPI processes, 1,000 input files will be created by the pre-processing tool partel. This high number of files can be a real issue on a cluster, as the cluster administrator usually limits the number of files per user. With the high number of files generated by partel, it can be a source of problem. Furthermore, operating systems have a limit on the number of files opened at the same time, and some users have noticed this limit which causes TELEMAC to crash.

The patch described in this section intends to reduce the number of input files to 2*X for TELEMAC2D, TELEMAC3D, TOMAWAC and SISYPHE, by making it independent from the number of MPI processes used.

To reduce the number of files generated by the pre-processing stage, there are mainly two solutions. The first one consists of removing the pre-processing tool and carry out the partitioning at the beginning of the simulations, so that every MPI process would access some parts of the original geometry file using MPI-IO. This solution would require to modify a lot of files in the codebase and add some complexity because of MPI-IO.

The second solution is to concatenate the partitioned geometry files into a single file. Instead of writing each partition in its own file, all of them are appended in a single file, as if they were in a queue. This is easy to do because partel is sequential and creates the partition in an ordered manner, from 1 to p. So when partel has finished to write the partition X in the file we know that we can safely append the (X+1)th in the same file. This is the solution we chose to implement.

1. *SERAFIN geometry files*

All the geometry mesh files created by partel are concatenated into a single file named XXXGEO-CONCAT (XXX stands for T3D/T2D/WAC/SIS). Neither information nor padding is added between the meshes. If you know at which byte the mesh begins and ends it is possible to read it as a normal serafin mesh. This information is stored in a second file called XXXGEO-INDEX. This index file contains the pairs of offsets on which the mesh begins and ends. The offsets are encoded in 64 bits integers and should be in big-endian (default for the TELEMAC system). The offsets are ordered from the mesh part 0 to p-1.

As an example, if the MPI process number 10 needs to read its mesh subdomain from the concatenated mesh file, it should retrieve the offsets which are in the position (10-1)*8*2+1 (two integers of 8 bytes on which we add one because the first position in the file is one, not zero). When these offsets are retrieved the *open_mesh_srf* subroutine uses the first one to set *pos_title* which corresponds to the beginning of the mesh. The other one is used to compute the number of timesteps.

2. *MED geometry files*

The MED file format allows the user to store several meshes into the same file, so we neither need to concatenate nor need the index file. In order to add a mesh into a file, we simply concatenate the original name of the mesh with the rank of the MPI process that uses it and then add it to the file with the normal MED function.

However, there are still some edge cases: TELEMAC uses what is called *parameters* in the MED file format to store some information which might be different for each mesh part, but MED only allows one set of *parameters* per file. Therefore, in order to store the *parameters* NPTIR for each mesh, its name has been concatenated with the rank of the MPI process. It would have been better to create one set of *parameters* per mesh part, but it is not possible.

3. *Boundary files (CLI) and PAR files*

These two files are ASCII files and for this reason very easy to concatenate. The boundary file uses the same kind of index file as the concatenate SERAFIN geometry file. The main difference is that the index is encoded as a 32 bits integer, but as it represents a line number and not a byte count, this should be sufficient. The main reason to do so is because it is then used in the HERMES module to perform a comparison against another 32 bits integer.

4. *Index files*

The index files for the GEO and CLI files are only a list of integers in 32 or 64 bits. They are binary files as they are not meant to be used by the end user. But if someone would like to take a look at it, it is easy do so by using the od command. For instance, to read the index of a concatenate SERAFIN file: *od --endian=big -t d8 T2DGEO-INDEX*

5. *Steering configuration files*

A new boolean keyword has been introduced "CONCATENATE PARTEL OUTPUT" to the dictionaries of the various modules in order to be used in the steering files. If concatenation is asked for, but there is only one process, no concatenation is performed. By default the concatenation is not activated.

6. *partel itself*

partel serial has been modified to ask the user whether it should create concatenated files or not. This has been done by adding another question to *homere_partel.f*. The python script *runcode.py* has also been modified to take the new keyword into account. It takes the value written in the steering file. If it is not present the default value is NO. This value is then added to the input file used by partel.

Some modifications have also been made in order to manage the PARAL and WEIRS files which are in read only mode. Previously all these files were copied and renamed, one for each MPI process, even if the process was only reading it. This has been changed, and the files are not copied anymore in such cases.

C. Intermediate conclusion

These modifications work fine and are about to be commited to the main repository. The added code only impacts some parts of the TELEMAC suite. The major changes are within the function *bief_open_files*. However, the complexity of the subroutine has not increased. The code was slightly refactored and might be even simpler than in previous versions. This new functionality is almost transparent for the user, the only change residing in adding a new keyword in the steering file if needed.

IV. Replacing the Fortran IO by MPI-IO to output the results

In the current version each MPI process outputs its result in its own file. This means that for p processes, p result files are generated. At the end of the computation the post-processing tool gretel is called to merge all the result files into a single one. As for the input, generating one file per process is a bad idea, it can quickly fill up the allowed quota given by a cluster administrator.

This section describes the proposed solution to reduce the number of result files to only one by making all the MPI processes write directly their results at the correct place in a unique result file. This implementation is available in the rainbowfish branch of the TELEMAC repository.

A. Explanation of the implementation

So far, the implementation only exists for the SERAFIN mesh files, so most of the modifications are made inside the *utils_serafin.f* file of the HERMES module. This file contains all the subroutines to read and write SERAFIN files. All of them use pure Fortran IOs. Writing the results is carried out sequentially, e.g. primarily some metadata are written at the beginning of the files and then, during the simulation, the result of each timestep is added at the end of the files. gretel is used to read all these results, reordering them and writing them into a single result file when the simulations are complete. By using MPI-IO it would be possible to write the results into a single file directly at the right offset while performing the simulations, hence removing the need for a post-processing tool.

As can be seen in the algorithm Algo 1, using MPI-IO is more complex than just changing Fortran IO for its MPI-IO counterpart, particulary if good performance is expected.

B. Performance considerations

Most high-end clusters use a parallel filesystem. For ARCHER it is Lustre. In order to get good performance on Lustre, frequently reading and writing operations should be avoided. In the original TELEMAC distribution, each process writes all the data to be dumped on the disk sequentially in a single file (the file index is the #processor minus one). As this step is sequential, only a few writings are required. But to get rid of the post-processing tool gretel another method is needed.

Furthermore, striping should be considered to reach good performance on Lustre, when using large files. On Lustre, each file can be divided transparently into several chunks. This is called striping. Each of these chunks can be modified in parallel. It is usually advised to stripe big files that are modified by several MPI process in order to achieve good performance (see [3]). Unfortunately, we did not have the chance to test different stripings, as it was corrupting the results produced by our implementation.

C. The algorithm step by step

In this section the algorithm Algo. 1 is explained. However its description is presented in a different order to the one it is actually executed in the code, as a way to more simply explain why the main operations are performed.

- Create a subcommunicator
- Create an MPI derived datatype for the whole file
- Write the header
- Repeat for each "graphic printout" until end of computation
  - Gather results
  - Order results
  - Write results

Algo. 1. Step to write the results with the MPI-IO version

1. Write results

Switching from Fortran IO to MPI-IO is not as easy as just replacing writing statements with *mpi_file_write*. By using *mpi_file_write* each MPI process writes independently from the others resulting in a lot of small outputs. To increase performance a collective writing is required, as for instance *mpi_file_write_all*. With this collective call all the writing processes synchronise and write at the same time. It the end the MPI runtime should be able to merge all the writes into a single one, or at least reduce the total number of writing operations.

Another optimisation is to overlap the writing by the simulations themselves. This can be done by using the nonblocking function *mpi_file_iwrite_all*. The only problem is that this function has been only added to the MPI 3.1 standard (2015), and is not supported by older MPI libraries.

2. Create an MPI derived datatype for the whole file

With the Fortran IO version (current distribution), each MPI process is writing parts of the final result in its own file sequentially. The new MPI-IO version is implemented to get rid of the post-processing tool gretel. This means that all the writers need to know where to write the data in the final file, and so each MPI process reorders its result. But this generates a non-contiguous array of data to dump to the disk. And this cannot be done in a single write by default.

The MPI standard provides a way to write efficiently data with this pattern. The solution is to create a model of the

whole file by using a MPI derived datatype. Each MPI process creates a type that represents the exact location of the part of the file it wants to modify. Then this type is used in the collective call of the *mpi_file_set_view* subroutine. With this new "view" of the file, each MPI process can write its result in a single write as if it was writing contiguous data. Since every process is doing a single write, they can do it with a collective write. This collective write should be optimised by the MPI runtime to obtain maximum efficiency.

3. Create a subcommunicator

A collective writing operation is needed to have good performance. But even with the collective writing routine, outputting the solutions can be still relatively slow when all the processes try to write at the same time. Profiling the code has shown that the more MPI processes try to write, the slower it will be. Profiling reports also identified as the cause the numerous small writing operations. This means that even with the collective routine, the MPI runtime is not able to merge the writing requests. To help it in this task, the number of writers has been reduced.

For each compute node one process is selected as the writer of the corresponding nod, usually the first one. All these writers are connected via a new communicator. When the program needs to write a result, all the non-writers of the compute node send their contribution to the writer. Afterwards all the writers start writing via a collective call using the new communicator, which is not the world communicator any longer. With this new way of writing the MPI runtime is now able to merge most of the small writes into bigger ones, which significantly improves the performance.

4. Gather the results

Since only one process per compute node writes the result, all the non-writers need to send their data to this one. This is done by a MPI communication before every writing. Even if it adds some complexity to the code, it is greatly beneficial because otherwise the collective call could not be used.

This step is performed by some gather functions, and is not very costly because the data do not move from a compute node to another one, hence all the operations use the memory of the same compute node.

5. Order the results

The MPI-IO version of the code needs to order the results. This step requires to be done before every writing operation and is actually carried out after every gathering step. Since there exists no sort function in the Fortran standard library, we have implemented a basic one (quicksort [8]), that might become a bottleneck in case of simulations using very large meshes, but this has not be observed yet in all the tests carried out in this work.

6. Other considerations

The SERAFIN files need to be written in big-endian format. The MPI standard provides a way to specify the endianess of the data to be written as an option in the *mpi_file_set_view*, native, internal or external32.

The external32 option provides a way to write in big-endian even when the processor is little-endian (most of the current existing processors), but it seems to be pourly implemented or not at all implemented in most MPI libraries. So we had to write a small set of functions to perform the conversion before writing the results.

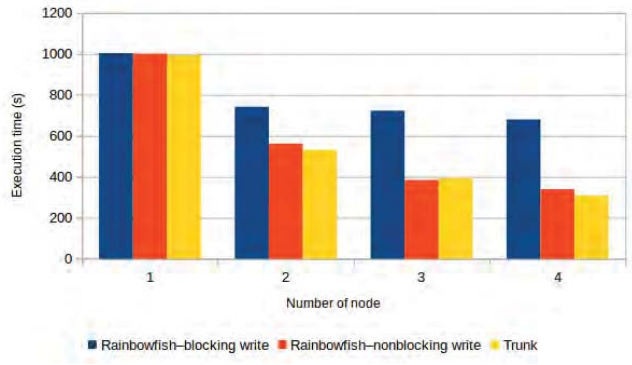

Figure 2. *Execution time comparison of the malpasset fine case with different writing methods and different numbers of compute nodes*

D. Performance comparison

The benchmarks have been carried out on ARCHER. Very different timings were obtained for a given simulation, and they showed a strong dependence on the load of the whole machine. In the worst case scenario a twofold increase in execution time could even be observed.

The performance comparison has been carried out in a way that all the measurements are recorded as closely to each other as possible. In order to do so, the number of executions was reduced, and the trust on the timings obtained was increased.

The test case used in the comparison is malpasset from the examples folder, with the case file *t2d_malpasset-fine.cas*. The mesh file has been refined twice via the script named *converter.py* to get a new mesh of 1,664,000 elements. The trunk code revision is 11,883 and the rainbowfish branch has been updated to that revision.

Figure 2 shows the comparison between two versions of MPI-IO, and Fortran IO from the trunk. In the rainbowfish blocking-write (BW) version, the program waits for the writings to be completed before continuing the computation. In contrast, the rainbowfish non-blocking write (NBW) version tries to overlap writings and computations.

Using a single compute node all the versions show the same performance. But when going for a second compute node, the BW version is outperformed by the NBW one. The NBW version and the trunk show similar performance for all tested numbers of compute nodes. Small differences may occur and result from the interference of the other users/processes running on ARCHER at the same time as the tests.

E. Intermediate conclusion

The MPI-IO implementation works fine and shows good performance for the SERAFIN file format. However, it adds a lot of new and complicated code lines to the HERMES module. It should also be noted that if the results produced by this new implementation are correct with the default options of the Lustre filesystem, they become wrong when changing the striping. It has not been possible so far, to identify where the problem comes from, whether it is because of our implementation, the Lustre filesystem or the MPI library.

Because of this issue the branch has not been merged, but this feature is at the moment available in the branch rainbowfish.

V. CONCLUDING REMARKS

In this paper an efficient way of reducing peak memory consumption in the pre-processing tool partel serial has been presented. This new implementation is aimed to be the default one for the next TELEMAC release.

Furthermore, a way to reduce the number of files generated by partel by concatenating them has been presented. This method works fine and should be pushed to the trunk in the coming weeks.

Finally, a method to reduce the number of result files generated by a run of TELEMAC by using MPI-IO has been presented. This method is still a work in progress and requires a lot of complex modifications to the codebase. Since the method used for partel has been proved successfully, more work should be devoted to investigate if this method could be used for the result files too instead of the here presented method using MPI-IO.

Appendix: Brief description of how a hash table works

In an array, every array index is mapped to an entry of the array. For instance, index 10 maps to cell 10 of the array. This data structure gives access to the cell in O(1). A downside is that if the index 10^6 has to be accessed but the previous one has not, it is still required to allocate an array of at least 10^6 elements. This is what happens in the original version of partel, the index being taken from a big set of indeces even if only a few of them are actually used.

A hash table is an array which indices are not necessary consecutive integers. More precisely a hash table is made of a standard array and a hash function.

The hash function is used to transform the non-consecutive indices given by the user to a smaller set of consecutive integers that map on the internal array of the hash table. An example of a very simple (but bad!) hash function would be h(k) = k mod n with k the index given by the user and n the size of the internal array. This function reduces the input integer enough to make it always fit in the table. The problem is that it outputs a lot of the same number for different inputs, which is called collision.

Even when using a good hash function, there will always be some collisions. There exist several techniques to manage these collisions. The one used in the newly developed version of partel is called linear probing. If two different inputs are mapped to the same cell of the table, we try to put the second one in the very next cell of the array; if this cell is already taken then we try the next one and so on until an empty cell can be found. The more we move further away from the original cell the more it takes time to write and read values in the table.

REFERENCES

[1] *Matthew Szudzik, Elegant Pairing, 2006, http://szudzik.com/ElegantPairing.pdf*

[2] https://github.com/google/farmhash

[3] http://www.archer.ac.uk/documentation/best-practice-guide/io.php

[4] Celis, P., Larson, P. A., & Munro, J. I. (1985, October). Robin hood hashing. In *Foundations of Computer Science, 1985., 26th Annual Symposium on* (pp. 281-288). IEEE.

[5] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

[6] https://doi.org/10.5281/zenodo.1323750

[7] http://valgrind.org/docs/manual/ms-manual.html

[8] Introduction to algorithms, second edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, MIT Press, 2001