Porting and Optimising TELEMAC-MASCARET for the OpenPOWER Ecosystem

Judicaël Grasset
STFC, Daresbury Laboratory
Warrington, United Kingdom
judicael.grasset@stfc.ac.uk

Yoann Audouin
EDF R&D
Chatou, France
yoann.audouin@edf.fr

Stephen Longshaw
STFC, Daresbury Laboratory
Warrington, United Kingdom
stephen.longshaw@stfc.ac.uk

Charles Moulinec
STFC, Daresbury Laboratory
Warrington, United Kingdom
charles.moulinec@stfc.ac.uk

David R. Emerson
STFC, Daresbury Laboratory
Warrington, United Kingdom
david.emerson@stfc.ac.uk

***Abstract*—TELEMAC-MASCARET is a suite of software for free-surface flow modelling. It is written in Fortran, parallelised with MPI and has been in development since the 1990s. This work aims to parallelise the code on the OpenPOWER architecture without heavily modifying its codebase. To do so the pragma-based programming directives from OpenMP and OpenACC have been tried on IBM POWER8 CPUs and NVIDIA GPUs. The results achieved for the wave propagation module of the suite on GPUs are promising and future works will be carried out on more challenging test cases.**

***Index Terms*—TELEMAC, POWER8, GPU, OPENMP, OPENACC**

I. INTRODUCTION

TELEMAC-MASCARET is an open-source suite of hydrodynamic solvers for free-surface flow modelling, originally developed by EDF R&D in the 1990s [1]. Development is now pursued through the TELEMAC-MASCARET Consortium. The software can be used to simulate 2-D or 3-D flows, sediment transport, water quality, wave propagation in coastal areas and rivers. More details on the possible applications and other capabilities can be found on the software's website [2].

At the moment TELEMAC-MASCARET is only parallelised with MPI, although attempts at hybrid parallelism have been tried in the past [3]. Improving the parallelisation and weak-scaling of TELEMAC-MASCARET would be useful for users who frequently do long time scale simulations.

Current computer trends favour the increase of the number of cores in a single processor and as shown by the current TOP500 list [4], this is alongside the addition of accelerators such as GPUs, combined with memory interconnects designed to reduce latency introduced by transferring data between different memory locations. It is therefore now important that TELEMAC-MASCARET is modified to take advantage of these CPUs and GPUs.

There are two key options when choosing how best to run on GPUs. Either going low-level and programme the kernel directly for GPUs with OpenCL or CUDA, or using pragma-based programming with OpenMP and OpenACC. The first option will give more control and usually more performance but it also means that a specific code has to be written and that two different versions of the same kernel has to be maintained. However, when using the pragma-based approach, algorithmic changes to code do not infer a re-write of the GPU kernel, with the bulk of the changes needed being the addition of pragmas around the existing code. This approach reduces the burden on those that maintain the original codebase and mean acceptance of changes is more likely. This work therefore concentrates on enabling multi-threaded CPU and GPU acceleration of portions of TELEMAC-MASCARET using a pragma approach.

This paper presents the use of OpenMP in order to reduce the number of MPI processes needed to utilise the CPUs in an OpenPOWER system, combined with the use of OpenACC and OpenMP to offload appropriate computations to available GPUs.

II. RELATED WORK

An attempt to use GPUs with TELEMAC-MASCARET has already been made in [3]. However the method used was completetely different from the one described in this article. Belaoura replaced the original matrix-vector product of TELEMAC-MASCARET with the one from the MAGMA library [5], which is able to offload it to GPU. The major problem they encountered was that the MAGMA library was not using the same matrix format. Doing the conversion before and after every matrix-vector product prevented any real-world performance improvement. This work shows how directly accelerating the existing data structures in TELEMAC-MASCARET allows significant gains to be made.

III. PORTING TO THE OPENPOWER ARCHITECTURE

The OpenPOWER foundation [6] is a consortium of entities working to provide an architecture revolving around the IBM POWER processors and accelerators. In this work the architecture used consists of POWER8 processors and NVIDIA GPUs. The processors are interfaced to the GPUs with NVLink instead of PCI-Express. NVLink a high-bandwidth proprietary interface developed by NVIDIA [7], is also used to enable GPU to GPU interconnection.

This work uses the UKRI Science and Technology Facilities Council (STFC) Paragon POWER8 cluster, maintained and run by the Hartree Centre [8] at Daresbury laboratory in Warrington in the UK. Each node of the cluster consists of 2 POWER8 CPUs, each of them with 8 physical cores (up to 8 hardware threads per core) and 4 NVIDIA P100 GPUs with NVLink 1.0 interconnects. Each P100 has 16GB of memory and the 2 POWER8 CPUs share 1TB of memory.

IV. Test case

In order to facilitate the evaluation of the OpenPOWER architecture a test case has been chosen in which most of the computational time is concentrated in a small part of the code and not spread accross a lot of different subroutines. Following benchmarks it was decided to use the *fetch_limited/tom_test6* case of the wave propagation module TOMAWAC of the TELEMAC-MASCARET suite. Preliminary benchmarks showed that about 95% of the execution time was spent in a single function called *qnlin3*. This function is short and is made of a four-level imbricated loop. As the original test-case mesh was very small, it was refined once in order to increase the computation time. This was achieved with STBTEL, a tool from the TELEMAC-MASCARET suite. The final mesh was made of 18,916 elements and 9,606 points. This test case is part of the official TELEMAC-MASCARET test suite and can be found freely with the source code.

All timings presented in this paper are for the whole duration of the program's execution and not only for the accelerated function. This ensures modifications are generally beneficial for users of the software and not only improvements visible in specific benchmarks.

V. Versions of software used

- TELEMAC-MASCARET V8P0R0 (revision 12565)
- Compiler IBM xlf 16.1.1.1
- Compiler GCC gfortran 8.2
- Compiler PGI pgfortran 18.10
- Library CUDA 9.2
- Library IBM Spectrum MPI 10.2.0

VI. Taking advantage of SMT

Each core of the POWER8 CPU is able to work at different levels of Simultaneous Multi-Threading, (SMT1, SMT2, SMT4, SMT8). This means that each core can execute more than one thread at the same time, e.g. two threads with SMT2. This functionality is comparable with the Hyperthreading technology of Intel processors. While Intel's Hyperthreading can only be used to run a maximum of two threads in parallel, a POWER8 core is able to run up to eight. Benchmarks have shown that TELEMAC-MASCARET does not benefit from the use of SMT8 (maybe because the memory bandwith is saturated, also SMT8 is not on par with SMT2 or SMT4 as it deactivates the CPU's instruction prefetcher [9]). Standard TELEMAC-MASCARET uses MPI parallelisation and is able to run on thousands of cores [10]. As shown in Table I, the code is able to benefit from using SMT to run MPI processes. It is always beneficial to use SMT2 and in some cases SMT4. This work therefore presents results using SMT1, SMT2 and SMT4.

TABLE I
ORIGINAL MPI VERSION. ONE MPI PROCESS PER ACTIVATED HARDWARE THREAD

PGI pgfortran	**SMT1**	**SMT2**	**SMT4**
1 node	1092s	857s	826s
2 nodes	569s	462s	452s
4 nodes	309s	258s	288s
8 nodes	174s	161s	169s
IBM xlf	**SMT1**	**SMT2**	**SMT4**
1 node	1264s	1018s	1019s
2 nodes	656s	552s	559s
4 nodes	356s	303s	329s
8 nodes	201s	181s	196s
GCC gfortran	**SMT1**	**SMT2**	**SMT4**
1 node	1388s	1034s	974s
2 nodes	639s	546s	526s
4 nodes	344s	295s	309s
8 nodes	193s	176s	182s

The problem with adding more MPI processes is that it increases the communication time for collective communication. Eventually it is likely that parts of the code will spend more time doing MPI communications than actually performing computation. To decrease this problem this work next looked at using OpenMP to parallelise the *qnlin3* subroutine and so reducing the number of MPI processes.

A. OpenMP

The *qnlin3* subroutine consists of a four level imbricated loop, with two arrays being updated in the most imbricated loop. OpenMP provides a set of directives to parallelise this kind of problem. In this case the best solution was to add a *parallel for* directive on top of the outermost loop. By doing so the processor is told to distribute the iterations of this to differents threads, and each of these threads will execute the whole of the three inner loops. Another point to take into consideration is the fact that several different iterations of the loops can modify the same index of the result arrays, therefore it is necessary to avoid this potential race condition. OpenMP offers two ways of doing this, either by declaring an instruction to be atomic or by using a reduction. Using atomic instructions is usually very costly on CPU, it is therefore preferable to use a reduction. One side-effect of using a reduction is that each thread needs to allocate a temporary array of the size of the original one, which significantly increases the total memory consumption.

Fig. 1 shows the execution time of this implementation with the IBM compiler. Each core executes an MPI process and a number of OpenMP threads, depending on the level of SMT. For instance, on one node with SMT4, 32 MPI processes are executed (16 per processor, 1 per core) and 64 OpenMP threads are executed (4 threads per MPI process). When compared to the original execution time (see Table I) it is clear that the implementation does not perform well.

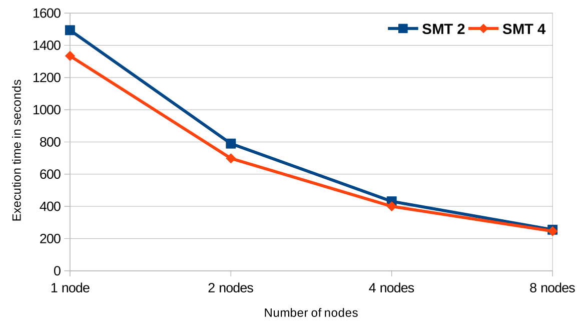

Fig. 1. MPI+OpenMP version. One MPI process per core and one OpenMP thread per activated hardware thread with the IBM compiler

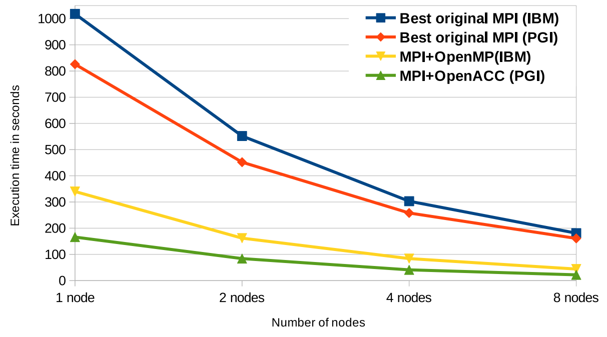

Fig. 2. Comparison of the best original MPI time against the modified MPI+OpenACC on GPUs version and MPI+OpenMP on GPUs version

In fact, in no case was it found to be better to replace MPI processes with OpenMP threads when using the IBM compiler. Executing one MPI process per processor then using all available cores and SMT for OpenMP threads was also tried, but the results were similar to the previous solution, showing no improvements against the pure MPI version with the IBM compiler. Some small tests have shown that there are some performance benefits when using the GCC compiler (see Table II) but the speedup is small (about 1.15x).

TABLE II
COMPARISON OF THE ORIGINAL MPI VERSION AND MODIFIED MPI+OPENMP VERSION ON 8 NODES WITH THE GCC GFORTRAN COMPILER

	SMT2	**SMT4**
Original MPI	176s	182s
New MPI+OpenMP	154s	160s

B. *Conclusion*

Testing and benchmarks have shown that, at least in this specific case with TELEMAC-MASCARET, the use of pure MPI achieves better performance on SMT enabled POWER8 processors than a hybrid MPI+OpenMP approach.

VII. TAKING ADVANTAGE OF GPUS

Following the current trend of adding or increasing the number of GPUs in HPC clusters, Paragon provides four NVIDIA P100 GPUs on each node. We have therefore investigated the possibilty of using these to increase the performance of TELEMAC-MASCARET.

A. *OpenACC*

OpenACC is an open standard set of directives to offload computations on GPUs, the standard is mainly developed by Cray and NVIDIA. While the test cluster used provides three compiler choices (GCC, IBM and PGI), only PGI appears to provide an efficient implementation of the OpenACC standard. The GCC compiler has an OpenACC implementation but it is still a work in progress, and IBM does not implement the OpenACC standard.

The OpenACC implementation for GPUs is quite similar to the OpenMP implementation for CPUs. OpenACC pragmas are used to collapse the four loops in the *qnlin3* subroutine used in the *fetch_limited/tom_test6* test-case and distribute the iterations on the available GPUs. The main difference of note for this work between OpenACC and OpenMP is that the version of OpenACC used (version 2.6) does not allow reduction on arrays (this functionality is available in OpenACC 2.7). To replace the reduction, atomic operations are used, in a pure CPU implementation this would be considered a bad approach as atomic instructions are typically slow but here GPU performance implications appear minimal. Using atomic operations also frees the code from creating and merging temporary arrays, leading to no notable increase in memory consumption. This is a welcome result as GPUs often have less memory available than CPUs.

In Fig. 2, results are shown for the OpenACC implementation compared to the original MPI version compiled with the PGI compiler. A notable improvement in execution time can be observed. On one node, the version running on GPUs is five times quicker than the original MPI version, on eight nodes it is seven times faster than the original. The OpenACC version was run on 4 MPI processes and 4 GPUs on each node, with each GPU being linked to an MPI process at the beginning of the program, which is the only process it then communicates with for the duration of its execution.

B. *OpenMP*

Since version 4.0, OpenMP has offered its own GPU offloading capabilities similar to those provided by OpenACC, again these are pragma-based. Even though the pragmas are completely different from OpenACC those used for offloading are almost functionally equivalent. As the PGI compiler used only supports OpenMP pragmas for CPU, the IBM compiler has been used to evaluate OpenMP GPU offloading performance.

Fig. 2 shows the results for the OpenMP offloading compared to the original MPI version, the two being compiled with the IBM compiler. It can be seen that there is still a notable acceleration when using the GPUs. On one node the version running on GPUs is three times faster than the

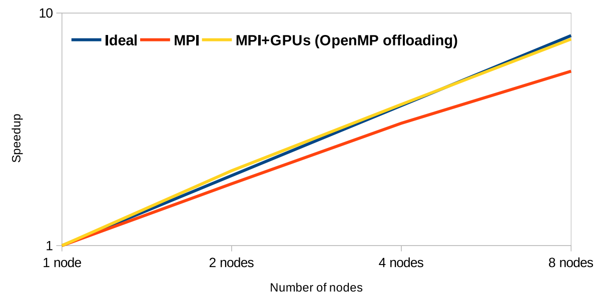

Fig. 3. Comparison speedup of the MPI+OpenMP version on GPUs against the original MPI version

original MPI version and it is four times faster on eight nodes. However the speedup achieved is smaller than the one achieved with OpenACC. In fact, the OpenMP version is about two times slower than the OpenACC version. This difference in performance could be attributed to having to use the IBM compiler rather than the PGI compiler used for the OpenACC tests as the IBM compiler produces slower, standard MPI code as it can be seen in Fig. 2.

C. Conclusion

It has been shown that it is possible, and beneficial, to use accelerators such as GPUs to accelerate some parts of TELEMAC-MASCARET, either by using OpenACC or by using OpenMP. The PGI compiler has been used for the OpenACC implementation and the IBM compiler for OpenMP. It would have been interesting in both case to try the GCC compiler (which should support offloading with either OpenACC or OpenMP) but significant results are yet to be generated, either because the implementation was very slow compared to the other compilers or because it was not working at all.

VIII. General Conclusion

This article first explored the use of a hybrid MPI+OpenMP implementation of the TOMAWAC portion of the TELEMAC-MASCARET suite of solvers for use on an OpenPOWER platform. However, results showed that the classical MPI-only implementation provided better utilisation, even on SMT-enabled POWER8 CPUs. In order to fully utilise the POWER8 platform, an evaluation of the use of GPU acceleration (by way of OpenACC and OpenMP pragmas) was also presented. It was found that it was possible for the test case used in this study to have a five to seven times speedup with PGI and OpenACC and a three to four times speedup with the IBM compiler and OpenMP in comparison to the original MPI version. Finally, as seen in Fig. 3 the scalability is also better than the original MPI version. This increase in performance will benefit users who are using similar cases, they will be able to either run their case quicker or to run it with the same execution time but use the acceleration to increase the accuracy of the simulation.

Future work will look to offload more modules of the suite to GPUs and use test cases provided by users who have computation time distributed across several subroutines. This work will be more complicated and may lead to smaller speedup figures because this will likely involve a larger number of discrete memory transfers between host and GPU. We will also evaluate how OpenMP and OpenACC offloading performs on the GCC 9 gfortran compiler.

IX. Acknowledgments

This work is supported by the Hartree Centre through the Innovation Return on Research (IROR) programme.

References

[1] GALLAND, Jean-Charles, GOUTAL, Nicole, HERVOUET, Jean-Michel. TELEMAC: A new numerical model for solving shallow water equations. Advances in Water Resources, 1991, vol. 14, no 3, p. 138-148.
[2] http://www.opentelemac.org/
[3] BELAOURA Hamza, Intégration de la bibliothèque MAGMA dans le système TELEMAC-MASCARET, Université de Versailles, Saint Quentin En Yvelines, Internship report
[4] https://www.top500.org/statistics/overtime/
[5] https://icl.utk.edu/magma/index.html
[6] https://openpowerfoundation.org/
[7] https://www.nvidia.com/en-gb/data-center/nvlink/
[8] https://www.hartree.stfc.ac.uk/Pages/home.aspx
[9] SINHAROY, Balaram, VAN NORSTRAND, J. A., EICKEMEYER, Richard J., et al. IBM POWER8 processor core microarchitecture. IBM Journal of Research and Development, 2015
[10] MOULINEC, Charles, DENIS, Christophe, PHAM, C.-T., et al. TELEMAC: An efficient hydrodynamics suite for massively parallel architectures. Computers & Fluids, 2011, vol. 51, no 1, p. 30-34.