Enhancing MPI+OpenMP Task based Applications for Heterogenous Architectures with GPU support

Manuel Ferat[1], Romain Pereira[2,4,5], Adrien Roussel[3,4], Patrick Carribault[3,4], Luiz-Angelo Steffenel[1], and Thierry Gautier[5]

[1] Université de Reims Champagne Ardenne, LICIIS, LRC DIGIT 51097 Reims, France
{manuel.ferat, luiz-angelo.steffenel}@univ-reims.fr
[2] CEA, DAM, DIF, F-91297 Arpajon, France
romain.pereira@cea.fr
[3] CEA, DAM, DIF, LRC DIGIT F-91297 Arpajon, France
{adrien.roussel, patrick.carribault}@cea.fr
[4] Université Paris-Saclay, CEA, Laboratoire en Informatique Haute Performance pour le Calcul et la simulation, 91680 Bruyères-le-Châtel, France
[5] Project Team AVALON INRIA, LIP, ENS-Lyon, Lyon, France
thierry.gautier@inrialpes.fr

Abstract. Heterogeneous supercomputers are widespread over HPC systems and programming efficient applications on these architectures is a challenge. Task-based programming models are a promising way to tackle this challenge. Since OpenMP 4.0 and 4.5, the *target* directives enable to offload pieces of code to GPUs and to express it as tasks with dependencies. Therefore, heterogeneous machines can be programmed using MPI+OpenMP(task+target) to exhibit a very high level of concurrent asynchronous operations for which data transfers, kernel executions, communications and CPU computations can be overlapped. Hence, it is possible to suspend tasks performing these asynchronous operations on the CPUs and to overlap their completion with another task execution. Suspended tasks can resume once the associated asynchronous event is completed in an opportunistic way at every scheduling point. We have integrated this feature into the MPC framework and validated it on a $AXPY$ microbenchmark and evaluated on a MPI+OpenMP(tasks) implementation of the LULESH proxy applications. The results show that we are able to improve asynchronism and the overall HPC performance, allowing applications to benefit from asynchronous execution on heterogeneous machines.

Keywords: OpenMP · GPU Computing · Distributed Application · Task programming

1 Introduction

Supercomputers race to the Exascale by significantly increasing the number of computing units per node and diversifying the types of computing resources

with accelerators (e.g., GPU). Numerical simulation applications need to handle computational complexity to remain efficient, but it is burdensome. Thus, parallel programming models require some evolutions to follow this trend. They attempt to leverage these hardware features in an almost transparent way while minimizing the stress put on application developers. Fine-grain parallelism with synchronization reductions is a solution to take advantage of many-core compute nodes.

The task programming model is thus an excellent alternative to the fork-join model. A task represents a piece of code and dependencies can be applied between tasks to represent partial order execution. Hence, a Task Dependency Graph (TDG) depicts the parallel section. Once a task is created, it can be executed instantly or deferred: programming makes asynchronous execution easier. Asynchronism is essential for the performance of applications designed for supercomputers in order to maintain a high level of parallelism and not to lose efficiency. A task-based programming model with dependencies has been added in OpenMP 4.0 [7] and has continued to grow in importance since then. In addition, supercomputers are composed of many compute nodes linked through a high-speed network. When coupling with MPI, the *de facto* standard used for distributed environment, the task programming model favors asynchronism to overlap communication with computation in distributed software [21, 22, 20].

OpenMP 4.0 introduces the `target` directives [18] to offload pieces of code to computing accelerators like GPUs. Since the advent of OpenMP 4.5 [19], `target` directives are explicit tasks with dependencies. This addition allows developers to write their entire application with a task-based approach homogeneously for heterogeneous architectures. Task representation may help overlapping data transfers with computation and then favor asynchronous execution between the host and the device.

Programming efficient codes exploiting all the parallelism provided by heterogeneous supercomputers remains challenging. While MPI+OpenMP programming model with tasks appears to be a good candidate for composing such applications, it still requires some effort at the runtime level to be efficient. Runtime stacking remains challenging, and the close collaboration between all the components is not granted. MPC [5] is a parallel framework that proposes an OpenMP implementation relying on a user-level thread scheduler. Recent work demonstrates that MPC can take advantage of interoperability between MPI and OpenMP(tasks) runtimes to enhance overall application performances [20]. In this paper, we propose the addition of GPU task support with the `target` directives in MPI+OpenMP(tasks) applications to enhance asynchronous executions thanks to user-space mechanisms. We have implemented this approach in the MPC framework and evaluated it on the LULESH mini application [12].

The contributions of this paper are the followings:

- integration of `target` directives in the MPC-OpenMP task based programming model,
- addition of user-space scheduling mechanisms to enhance asynchronous GPU executions,

– porting and evaluating MPI+OpenMP(tasks) LULESH mini application for heterogeneous architectures.

The paper is organized as follows. Section 2 presents related work in task-based programming models for distributed heterogeneous applications. Section 3 details the port of OpenMP `target` directives and its integration into the MPC framework to enhance asynchronism thanks to the GPU task suspension support. Section 4 sketches the porting of MPI+OpenMP(tasks) with GPU support for some applications. Section 4.2 presents the performance evaluation of our work. Finally, Section 5 concludes this work and depicts future work.

2 Related Work

Heterogeneous programming gained importance as GPU computing accelerators are widespread in HPC systems. Tasked-based runtime systems have shown that task programming is well designed for the composition of heterogeneous applications. StarPU [1], X-Kaapi [8] and OmpSs [2] enable developers to compose an application with various types of tasks that can be performed both on the GPU and the CPU. These runtime systems can overlap data transfers with computation through data prefetching techniques thanks to CUDA streams. Legion [3, 9] is also a task-based runtime designed for distributed machines with heterogeneous nodes. More recently, CudaGraph [14] enables developers to write or capture GPU operations only and organizes them into graphs to reduce kernel launch overheads of CUDA. OpenMP [4, 18] target constructs have been introduced in the specification version 4.0. OpenMP 4.5 [19] defines target constructions as tasks, and task programming seems to be growing for the future OpenMP 6.0. OpenMP standard guarantees the perpetuation of simulation codes by integrating heterogeneous task programming [24].

Offloading data and kernels on GPU are blocking operations by default but can be transformed into asynchronous ones to gain parallel efficiency. In the same way that MPI communications can block the execution of the tasks [23], synchronous GPU operations may degrade the performances. In 2015, MPI+ULT (User Level Thread, Argobots) [16] proposed to run MPI code within a user-level thread and make it yield whenever an MPI communication blocks. MPI Detach [21] advocates programming through continuations. It enables the addition of asynchronous callbacks on communication completion but implies heavy code restructuration. TAMPI [22] and MPC [20] propose to transform synchronous to asynchronous MPI calls finely nested into OpenMP tasks. These works tackle the interoperability issue between MPI and OpenMP tasks but do not consider heterogeneous applications. As OpenMP target constructs are now explicit tasks, these approaches may be applied to GPU task programming.

In [25], several approaches are presented to overlap GPU operations with computations thanks to OpenMP target constructions. They proposed to run asynchronous target tasks within dedicated threads, which are preempted by blocking operations. These threads are called Hidden Helper Threads (HTT) and are implemented as *kernel* threads, hence delegating scheduling decisions to

the operating system. Standardization of this approach was suggested in [15]. This paper presents a different design for asynchronous GPU calls overlapping through user-space cooperative scheduling. This approach targets efficient MPI+OpenMP(tasks) applications for heterogeneous supercomputers.

3 Targets with Asynchronous Tasks

The main goal of asynchronous offloading is to overlap accelerator operations with useful work on the CPUs. With the task programming model, this is achieved through task switching during asynchronous operation progression.

3.1 Cooperative Target Task Design

The general idea of the Cooperative Target Task Design is to have the target tasks make asynchronous calls and explicitly preempt right before synchronization points without the operating system scheduler intervening. Polling functions ensure asynchronous events progression. On completion, the preempted task is unblocked and may resume. This design strictly distinguishes *threads* and *tasks* as depicted in the Figure 1.

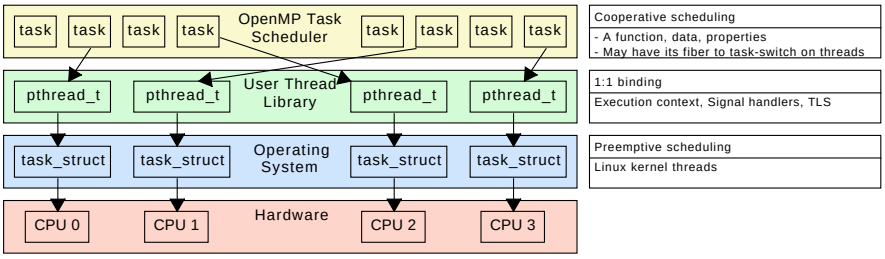


Fig. 1. Fiber task design : from physical core to user-space tasks

Threads are standard OpenMP threads represented as `pthreads` and mapped 1:1 with kernel threads and physical cores. They have their own user and kernel space execution *context* made of a stack and registers copy. They also have their own signal handlers, file descriptor table, and Thread Local Storage (TLS).

Tasks are standard OpenMP tasks made of a function, some data, and properties. In addition, they may also have their own user-space execution context known as a *fiber* [13]. A task can run on top of a thread on its own fiber if it has one or directly on the thread fiber otherwise. With the fiber capability, a task can pause itself explicitly at any time of its execution and may resume on any

threads (if untied) once it unblocks. Yet, asynchronous events still have to be polled at some point. In our approach, this is done opportunistically on every scheduling point defined in the OpenMP Specification.

3.2 OpenMP Target in MPC

Multi Processor Computing (MPC) [5] implements its own OpenMP runtime through LLVM and GCC Application Binary Interface (ABI). It also improves standard OpenMP task capabilities with fibers through Linux `<ucontext.h>` interface for cooperative scheduling. Previous work has been done to finely nest MPI communications into OpenMP(task) [20]. This suspend/resume task mechanism allows generating asynchronism that can be usefully exploited by an application to overlap MPI communications with computations. Implementing OpenMP targets with the introduced design now only becomes a question of Application Binary Interface (ABI) support and tasks cooperativity.

Implementation To add target directive support to MPC, we chose to use the LLVM *libomptarget* library because it is built as a sub-module of the LLVM OpenMP project. The OpenMP Target part is slightly coupled with the rest of the project, which will ease its integration with another OpenMP implementation such as MPC-OpenMP.

MPC implements the LLVM ABI which makes possible to compile with *clang* and to link with both the MPC-OpenMP runtime and the LLVM `libomptarget` library. To make this combination functional, we had to modify several entry points in the MPC-OpenMP runtime. First, when starting a program using LLVM's `libomptarget`, even before the main execution, the `libomptarget` library is initialized through `__kmpc_get_target_offload` ABI which was implemented in the MPC-OpenMP runtime. It reads the `OMP_TARGET_OFFLOAD` environnement variable from the OpenMP runtime and passes it to `libomptarget`. Then, the device's specific libraries are loaded. We also had to implement the support of the `omp_get_default_device` and `omp_is_initial_device` OpenMP API in MPC-OpenMP.

Finally, to enable the expression of target directives as tasks with dependencies, we added the support of the `__kmpc_omp_target_task_alloc` ABI. This creates a task that encapsulates the incoming target region, generating a standard and opaque task for the OpenMP runtime. The internal task function points to an entry point in the `libomptarget` responsible of starting and completing the asynchronous GPU operation. The completion of GPU operations implies synchronizations that end up blocking threads. Hence, the LLVM OpenMP runtime executes asynchronous target tasks on dedicated Hidden Helper Threads (HHT) [25] implemented as *kernel* threads. Thus, the operating system can preempt threads blocking on GPU operations, and Standard OpenMP threads can be rescheduled onto physical cores to progress other tasks in parallel.

In MPC-OpenMP, we decided to implement the Cooperative Target Task design instead of the HHT design for its user-space scheduling flexibility. Still, the target tasks end up synchronizing at some points blocking threads and cores.

Enabling Asynchronism through Cooperativity In practice, synchronizations happens at the end of target execution with a CUDA stream synchronization within the `libomptarget`. Without cooperativity, the target tasks ends up retaining the physical core. To tackle this issue, we patched the LLVM `libomptarget` CUDA Runtime Library (RTL) [6] to modify the implementation of `DeviceRTLTy.synchronize`. It now inserts a CUDA Stream progression polling function into MPC relying on `cudaStreamQuery` and explicitely yields until the progression is completed. Moreover, programmers can also decide whether target tasks should have their own fibers and mark them as `untied`, so that they may resume on any thread at any scheduling point. Hence, MPC threads will not block cores and the thread will overlap the stream progression with useful computation through pure user-space task switches. This approach enables fully asynchronous support of OpenMP targets in MPC-OpenMP.

3.3 State-of-the-Art Comparison

Microbenchmark In order to compare our approach with existing solutions, we extended the microbenchmarks from [25]. The Listing 1.2 depicts our microbenchmark B5. The x and y vectors are of size $n.T$ with $(n, T) \in \mathbb{N}^2$. A single thread produces $T \in \mathbb{N}$ host-to-device data transfers (line 5), computation kernels (line 7) and device-to-host data transfers (line 10). These operations are not grouped in one construction to ensure the most asynchronism. The target tasks triplets are ordered with dependencies, and up to T triplets can be consumed concurrently by any threads. Each data transfers complexity is $O(n)$ bytes while computation kernels time complexity is $O(n^2)$ as shown in the Listing 1.1 and 1.2.

Listing 1.1. daxpy-like function

```
1 # define daxpy(A, X, Y, I0, IF)              \
2     for (uint64_t I = I0 ; I < IF ; ++I)     \
3         for (uint64_t J = 0 ; J <= I ; ++J) \
4             Y[I] = Y[I] + A * X[J];
```

Listing 1.2. Target microbenchmark B5

```
1  void B5(double a, double * x, double * y, int T, int n) {
2      # pragma omp parallel
3      # pragma omp single
4      for (int t = 0 ; t < T ; ++t) {
5          # pragma omp target update to(y[t*n:n]) nowait depend(inout: y[t*n])
6
7          # pragma omp target teams distribute parallel for nowait
               depend(inout: y[t*n])
8          daxpy(a, x, y, t*n, n);
9
10         # pragma omp target update from(y[t*n:n]) nowait depend(inout:
               y[t*n])
11     }
12 }
```

[6] https://gitlab.inria.fr/ropereir/iwomp2022

Environment Our experiments run onto nodes made of two AMD EPYC 7H12 64-core processors. Each processor has 4 NUMA domains with 32GB memory and 16 cores. Moreover, each processor has 2 Nvidia A100 GPUs connected to their NUMA domain 0 and 2. We use a compact threads pinning on NUMA domains 0 or 2 and we only use one GPU at a time. We set $T = 64$ and vary n in $\{2^4, 2^5, ..., 2^{16}\}$. Each point is the median of 5 runs, and the error bars represent extremums. We used LLVM release 14.x suite, Nvidia/PGI 22.2 suite and MPC-OpenMP runtime with LLVM release 14.x compiler and the patched libomptarget. Every software was compiled with $O3$ optimizations enabled.

Results Each run was wrapped into NVIDIA Nsight Systems tracing and each runtime showed a similar execution time on the compute kernel and the two data transfers, which means that the observed performance differences mostly come from task scheduling. Figure 2 depicts the performances varying the number of threads. For each runtime, the number of threads corresponds to the parallel region, but for LLVM, it also corresponds to the number of Hidden Helper Threads (HHT) using `LIBOMP_NUM_HIDDEN_HELPER_THREADS` environment variable. The performances are represented as the speedup relative to LLVM with 8 OpenMP threads and 8 HHT. The best performance reaches about 625 GFlop/s for $n = 2^{16}$ with both LLVM using 16 threads and MPC using 4 threads.

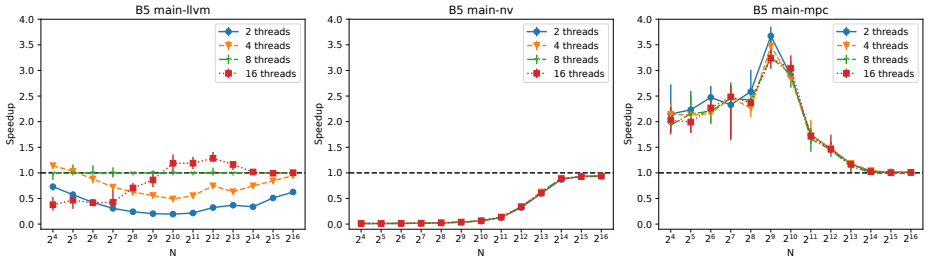


Fig. 2. OpenMP runtimes speedup relative to LLVM with 8 hidden helper threads

LLVM We observe that the number of HHT can significantly impact performances regarding the state of art performances. While the 8-threads default configuration seems a reasonable compromise on average. We still observed up to 1.14 speedup on fine-grain using 4 threads and 1.3 speedup using 16 threads on medium grain.

Nvidia/PGI Our installation is low-performing on fine-grain offloading (1% to 5% of LLVM performances). For coarser grains, performances improve but are still about 8% slower. We also observe that the number of threads does not impact performances. CUDA driver calls are only performed on a single thread, making the GPU underloaded on the fine-grain configurations.

MPC Cooperative task scheduling on target tasks significantly improve performances on fine-grain offloading. The performances are converging to LLVM-OpenMP for coarser grains, likely because the GPU cores are becoming overloaded. Another benefit of the user-space scheduling approach is the small performance variation with the number of threads. As LLVM results show, performances vary by a factor up to 6 depending on the number of threads, on $n = 2^{10}$ between 2 and 16 threads. With MPC, this variation is maximum between 2 and 16 threads on $n = 2^9$ with a factor of 1.2.

Conclusion This microbenchmark shows that our cooperative target tasks approach efficiently schedules asynchronous GPU operations. Moreover, it relieves the burden of hidden helper thread configuration from users - which can significantly impact performances on LLVM - by delegating the scheduling decisions from the operating system to the OpenMP task scheduler. Now comes the question of performance on real-world applications.

4 Heterogeneous and Hybrid Task-Based LULESH

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) is a proxy application representing the core of hydrodynamics in real-world applications. It has been widely studied under different programming models but not under fully asynchronous, hybrid, and heterogeneous task programming. Hence, we decided to port this application to MPI+OpenMP(tasks+target) standards and evaluate our scheduling approach.

4.1 Porting

Tasking Original application `parallel for` loops were transformed to tasks generating loops with dependencies in a single-producer/multi-consumer scheme. Thus, we defined the parameter `-te` as the number of tasks decomposing a loop.

Heterogeneity We separate computation offloading through the target construct from data transfers between host to device with the target enter/exit data directives. Order of execution is guaranteed through data dependencies which allows independant operations to operate concurrently. In practice, we offloaded the first loop of `IntegrateStressForElems` and the `CalcKinematicsForElems` loops because they depend on host-to-device data transfers that can be overlapped with other CPU computation such as `CalcAccelerationForNodes` loops for instance. Moreover, we also switch the standard `malloc` memory allocator to `cudaMallocHost` to allocate pinned memory. Otherwise, the device to host memory transfers cannot be overlapped with computation on the CPUs. [7]. To adjust GPU tasks granularity, GPU tasks were made "super-tasks" merging several original CPU tasks. The `-st` parameter controls the number of original CPU tasks composing a single GPU super-task.

[7] https://docs.nvidia.com/cuda/cuda-driver-api/api-sync-behavior.html

Hybridation MPI communications were finely nested within OpenMP tasks and ordered with dependencies to support distributed computing. We are using the MPC-OpenMP interoperability layer presented in [20] to automatically overlap blocking calls through user-space task switches. Hence, tasks with MPI calls can be inserted into the Task Dependency Graph (TDG), just like regular tasks.

Optimizations We backported the *global allocation of temporary work arrays* optimization proposed in [10] in the baseline version and our task-based version. This optimization consists of preallocating reusable memory buffers instead of reallocating them on every iteration in the original code.

Representativity, correctness, standard Original authors provide guidelines on porting to keep the code representative of ALE3D. Moreover, authors also provide minimal cases results to ensure the correctness of various studies [11, 12] When porting the application, we fully respected those guidelines and ensure the correctness of our approach. Moreover, the source code of the application and our MPC-OpenMP runtime are available online [8]. Note that our version is not fully standard compliant. In particular, we are using some specific MPC-OpenMP runtime call to bypass restrictions on `depobj` and `iterators`, and the lack of support for the `inoutset` dependency type by compilers.

4.2 Evaluation

The upcoming experiments aims to demonstrate that the hybrid and heterogeneous task-based version of LULESH manage to overlap GPU/MPI asynchronous operations with computation within a unified OpenMP task scheduler.

Experimental setup In this section, we always run 5 times the application and presents the median and extremum values. We are using the same AMD+A100 nodes positionning MPI ranks per 16-core NUMA domain with 1 GPU. We run our experiments with the same software stack presented in the Section 3.3. The tasks versions always run with MPC-OpenMP and the patched `libomptarget` presented in the Section 3.2. The problem dimension was set to `-s=264` which fills 80% of the NUMA domain memory capacity and 20% of the GPU memory capacity. The parameters `-te` and `-st` represents the loop cutting into tasks and must be evaluated for fine tuning.

Loop cutting into tasks Our first experiment consisted in determining the best value for `-te` on the given problem dimension. We fixed `-st=1` and disable GPU offloading, so that every tasks run onto CPUs. We varied the parameter `-te` in $[8, 2048]$ on single rank runs and `-i=32` iterations. This parameter changes both the parallelism and the tasks grain: the more tasks, the more parallelism,

[8] https://gitlab.inria.fr/ropereir/iwomp2022

but the finer the grain. The best performances were found for `-te=1280` so this is the value we used in next experiments.

In a second phase, we enabled GPU offloading and varied `-st` in $[1, 1280]$. We compared the performance results obtained for each task cutting with the OpenMP `parallel-for` baseline version and our task-based version disabling GPU offloading . The results are depicted in the Figure 3 and show that the lowest execution time for `-st=4` and `-st=8`. The right side is a zoom of the left side to see the lowest execution time better. First, we observe that the task version

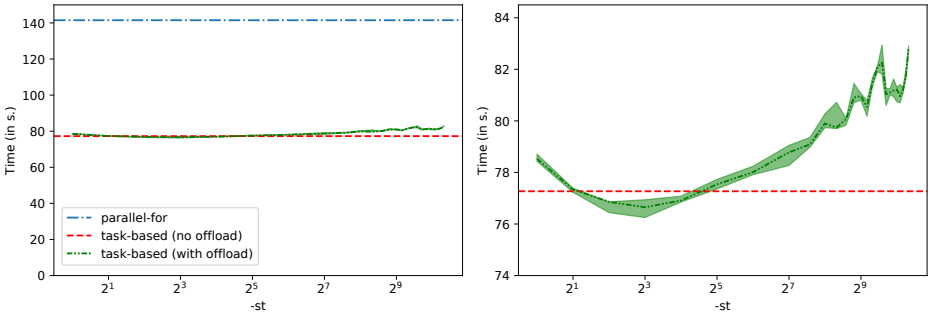


Fig. 3. Task merging into super-tasks study

significantly improves performances over the reference parallel-for version. After investigating with `perf` [6], we found out that this important performance gain comes from the data cache reuse. We are using a LIFO scheduling strategy that favors the execution of the tasks direct successors. This strategy ends up following the data movements, which significantly reduce cache misses and the process pipeline stalls, improving the number of instructions per cycle from 0.52 to 0.90. This phenomenon is called *work time inflation* and was already shown in [17]. Regarding the GPU offloading, we measured the best performances with `-st=4`. We do not observe a significant performance gain with the GPU offloading and tasks fibers enabled even with this finely tuned parameter. We have looked at some possible explanations.

Overlapping and Scheduling The Figure 4 depicts the gantt chart on an instance of execution for the same problem size. The highlighted task is a GPU kernel from the `CalcKinematicsForElem` loop. As you can see, the target task starts

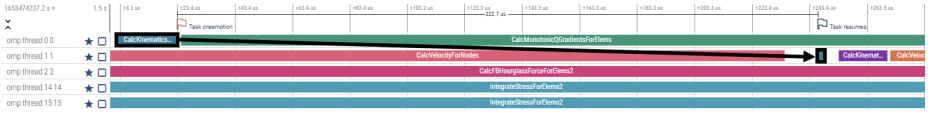


Fig. 4. LULESH CalcKinematicsForElem offloading overlap

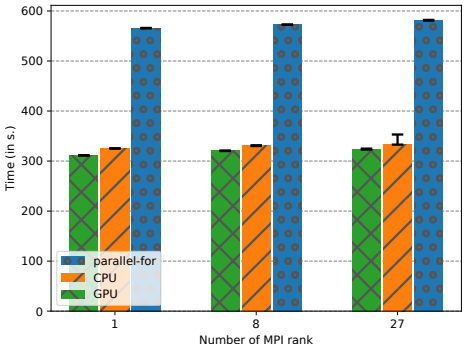
and blocks on thread 0 to resume later on thread 1. Thread 0 overlaps the kernel execution by switching to a `CalcMonotonicQGradientsForElems` task. While the overlapping mechanism is working as expected, we found a few cases where the threads have no more ready tasks on the CPU-side to overlap asynchronous operations leading to short idle periods. Yet, we measured less than 1% of idle time overall, which removes this hypothesis to explain the small performance gain.

Workload, arithmetic intensity, data transfers Using MPC tracing, we measured that the offloaded loops represent 15% of the overall work when running on CPUs. This limits the amount of work available to the GPU, which ends up being underused. Using NVIDIA Nsight System tracing, we measured 0.7 s. spent by the GPU in the computation kernels and 18.9 s. spent on the data transfers. This also shows the short *arithmetic intensity* of the offloaded kernel. This may perturbate the task execution on CPUs leading to *work time inflation*. Still, the host and device memory transfers are mostly overlapped, as depicted in the Figure 4 so we would have expected higher performances.

Work time inflation NVIDIA tools reported 512,000 data transfers of 0.38MB on average that our OpenMP runtime blindly schedules as normal tasks. We decided to measure the impact of those many data transfers on CPU performances, by running the task with GPU offloading under `perf`. We measured that the offloading version has 14% fewer instructions than the pure-CPU version - which corresponds to the 15% CPU work offloaded to GPU we measured previously with MPC. But more importantly, `perf` shows that the instruction/cycle drops from 0.90 to 0.62 and MPC shows a +12% work time inflation on the CPU tasks when enabling the offloading. In other words, the same tasks running on CPUs take 12% more time when we offload some work to the GPU. We do not yet have an exact explanation for this.

Conclusion To conclude this preliminary experiments, setting `-te=1280` and `-st=4` enables efficient loop cutting into tasks for the given problem size. Asynchronous tasks execution between host and device increases the performance of the LULESH application. However, this gain is compensated by the work time inflation on CPU tasks and would need more investigation.

Weak-scaling This last experiment is a weak-scaling from 1 to 27 processes distributed over 7 nodes. We are running the same problem but increasing the number of iterations from 32 to 128. We used Open MPI 4.0.5 with MPC-OpenMP interoperability library. Every version of the application scales very well to 27 MPI ranks. Compared to the baseline version, both task versions show a significant speedup, which comes from better cache reuse, as shown in the previous experiment. We also observe a slight performance gain when offloading work to the GPU. This result shows that (almost) standard OpenMP asynchronous hybrid and heterogeneous task-based programming model enables scalable applications.



	Median time (in s.)		
ranks	parallel-for	no offloading	offloading
1	565.84	325.02	309.14
8	572.45	331.09	317.91
27	581.26	333.98	318.83

Fig. 5. LULESH weak-scaling from 1 to 27 GPUs

5 Conclusion and Perspectives

Heterogeneous supercomputers achieve high performance but add a new level of asynchronicity. It makes the programming of efficient HPC applications challenging. OpenMP task-based programming is promising to compose every parallelism level. However, the overlapping of data transfers and GPU offloading with computations on CPUs are only possible if task suspension is efficiently supported. In this paper, we propose a user-space cooperative target task design to enable the execution of asynchronous and heterogeneous applications on distributed machines.

We have integrated this mechanism into the MPC framework and the LLVM `libomptarget` for the GPU offloading support. We show that the Cooperative Target Task design offers more flexibility to the task scheduler than the Hidden Helper Threads (HHT) state-of-art design which delegates scheduling decisions to the operating system. We also have ported the LULESH proxy applications to the MPI+OpenMP(tasks) programming standards for heterogeneous and distributed architectures. The results show that performances can benefit from asynchronous executions. However, the gain observed are hidden by work time inflation of CPU tasks, and, in future works, it requires more investigation to understand this behavior.

Currently, we only support the asynchronous target task execution on NVIDIA GPUs. As a perspective, we plan to add support for other GPU vendors (AMD, Intel). It will likely be through patching other `libomptarget` RTL plugin implementations to support asynchronous offloading in a portable way. In this paper, the asynchronous target task design relies on a few non-standard runtime capabilities. In particular, MPC-OpenMP tasks can run onto their own *fiber*, block / unblock explicitly and resume on any threads. The current standard `taskyield` directive and `untied` clause does not give such guarantees to the programmer.

Acknowledgements

Present results have been obtained within the frame of DIGIT, Contractual Research Laboratory between CEA, CEA/DAM-Île de France center and the University of Reims Champagne Ardenne.

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2), 187–198 (2011). https://doi.org/10.1002/cpe.1631
2. Ayguadé, E., Badia, R.M., Igual, F.D., Labarta, J., Mayo, R., Quintana-Ortí, E.S.: An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In: Sips, H., Epema, D., Lin, H.X. (eds.) Euro-Par 2009 Parallel Processing. pp. 851–862. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
3. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 1–11 (2012). https://doi.org/10.1109/SC.2012.71
4. Beyer, J.C., Stotzer, E.J., Hart, A., de Supinski, B.R.: OpenMP for Accelerators. In: Proceedings of the 7th International Conference on OpenMP in the Petascale Era. p. 108–121. IWOMP'11, Springer-Verlag, Berlin, Heidelberg (2011)
5. Carribault, P., Pérache, M., Jourdren, H.: Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More. pp. 1–14. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
6. De Melo, A.C.: The new linux'perf'tools. In: Slides from Linux Kongress. vol. 18, pp. 1–42 (2010)
7. Duran, A., Ferrer, R., Ayguadé, E., Badia, R.M., Labarta, J.: A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks. Int. J. Parallel Program. **37**(3), 292–305 (jun 2009). https://doi.org/10.1007/s10766-009-0101-1
8. Gautier, T., Lementec, F., Faucher, V., Raffin, B.: X-Kaapi: a Multi Paradigm Runtime for Multicore Architectures. In: Workshop P2S2 in conjunction of ICPP. p. 16. Lyon, France (Oct 2013), https://hal.inria.fr/hal-00727827
9. Jia, Z., Kwon, Y., Shipman, G., McCormick, P., Erez, M., Aiken, A.: A Distributed Multi-GPU System for Fast Graph Processing. Proc. VLDB Endow. **11**(3), 297–310 (nov 2017). https://doi.org/10.14778/3157794.3157799
10. Karlin, I., McGraw, J., Keasler, J., Still, B.: Tuning the LULESH Mini-app for Current and Future Hardware (2013), https://www.osti.gov/biblio/1070167
11. Karlin, I.: LULESH Programming Model and Performance Ports Overview (2012)
12. Karlin, I., Keasler, J., Neely, R.: LULESH 2.0 Updates and Changes. Tech. Rep. LLNL-TR-641973 (August 2013)
13. Kowalke, O.: Distinguishing coroutines and fibers (2014)
14. Lin, D.L., Huang, T.W.: Efficient GPU Computation Using Task Graph Parallelism. In: Sousa, L., Roma, N., Tomás, P. (eds.) Euro-Par 2021: Parallel Processing. pp. 435–450. Springer International Publishing, Cham (2021)

15. Lopez, V., Criado, J., Peñacoba, R., Ferrer, R., Teruel, X., Garcia-Gasulla, M.: An openmp free agent threads implementation. In: McIntosh-Smith, S., de Supinski, B.R., Klinkenberg, J. (eds.) OpenMP: Enabling Massive Node-Level Parallelism. pp. 211–225. Springer International Publishing, Cham (2021)
16. Lu, H., Seo, S., Balaji, P.: MPI+ULT: Overlapping communication and computation with user-level threads. pp. 444–454 (08 2015). https://doi.org/10.1109/HPCC-CSS-ICESS.2015.82
17. Olivier, S.L., de Supinski, B.R., Schulz, M., Prins, J.F.: Characterizing and mitigating work time inflation in task parallel programs. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '12, IEEE Computer Society Press, Washington, DC, USA (2012)
18. OpenMP Architecture Review Board: OpenMP Application Program Interface Version 4.0 (2013), http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
19. OpenMP Architecture Review Board: OpenMP Application Program Interface Version 4.5 (2015), https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
20. Pereira, R., Roussel, A., Carribault, P., Gautier, T.: Communication-Aware Task Scheduling Strategy in Hybrid MPI+OpenMP Applications. In: McIntosh-Smith, S., de Supinski, B.R., Klinkenberg, J. (eds.) OpenMP: Enabling Massive Node-Level Parallelism. pp. 197–210. Springer International Publishing, Cham (2021)
21. Protze, J., Hermanns, M.A., Demiralp, A., Müller, M.S., Kuhlen, T.: MPI Detach - Asynchronous Local Completion. In: 27th European MPI Users' Group Meeting. p. 71–80. EuroMPI/USA '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3416315.3416323
22. Sala, K., Teruel, X., Pérez, J., Peña, A., Beltran, V., Labarta, J.: Integrating Blocking and Non-Blocking MPI Primitives with Task-Based Programming Models. Parallel Computing **85**, 153–166 (07 2019). https://doi.org/10.1016/j.parco.2018.12.008
23. Schuchart, J., Tsugane, K., Gracia, J., Sato, M.: The Impact of Taskyield on the Design of Tasks Communicating Through MPI. In: de Supinski, B.R., Valero-Lara, P., Martorell, X., Mateo Bellido, S., Labarta, J. (eds.) Evolving OpenMP for Evolving Architectures. pp. 3–17. Springer International Publishing, Cham (2018)
24. de Supinski, B.R., Scogland, T.R.W., Duran, A., Klemm, M., Bellido, S.M., Olivier, S.L., Terboven, C., Mattson, T.G.: The Ongoing Evolution of OpenMP. Proceedings of the IEEE **106**(11), 2004–2019 (2018). https://doi.org/10.1109/JPROC.2018.2853600
25. Tian, S., Doerfert, J., Chapman, B.: Concurrent Execution of Deferred OpenMP Target Tasks with Hidden Helper Threads. In: Chapman, B., Moreira, J. (eds.) Languages and Compilers for Parallel Computing. pp. 41–56. Springer International Publishing, Cham (2022)